UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA U
NOVOM SADU

Dušan Okanović

Model adaptivnog sistema za praćenje i predikciju rada distribuiranih aplikacija

- doktorska disertacija -

Novi Sad 2012

Predgovor

Pored izvršavanja osnovnih funkcionalnosti, softver mora da ispunjava i određene nefunkcionalne zahteve. U nefunkcionalne zahteve spadaju npr. dostupnost, vreme odziva, sigurnost i zauzetost resursa. Ispunjavanje nefunkcionalnih zahteva, koji predstavljaju nivo servisa, se prati tokom celog životnog ciklusa softvera. Praćenje rada aplikacija i predikcije njihovog rada spada u oblast upravljanja nivoom servisa.

Podaci dobijeni praćenjem mogu da se koriste za:

- sticanje slike o performansama aplikacije,
- utvrđivanje poštovanja ugovora o nivou servisa (SLA),
- predikciju ponašanja aplikacije, kao i
- planiranje održavanja aplikacije.

U procesu razvoja, za proveru ponašanja softvera koriste se alati koji obično dodaju značajno opterećenje (eng. *overhead*). Ovo opterećenje nije bitno u toku razvoja, ali za krajnje korisnike je neprihvatljivo.

U fazi testiranja nemoguće je testirati uticaj okruženja aplikacije i opterećenje od strane krajnjih korisnika. Često se pojavljuju problemi koji nisu primećeni u toku faze testiranja.

Da bi bili sigurni da aplikacija zadovoljava sve nefunkcionalne zadatke koji su postavljeni pred nju, tj. da zadovoljava parametre definisane ugovorom o nivou servisa, neophodno je da se rad softvera kontinuirano prati. Ako se javi nekakvo odstupanje od zadatih parametara, kontinuirano praćenje može da omogući otkrivanje uzroka.

Na osnovu sakupljenih podataka, moguće je predvideti dalje ponašanje aplikacije i izvršiti izbor daljih akcija. Ovo je značajno npr. za planiranje proširenja kapaciteta softverskog sistema, da bi on i održao zadati nivo performansi u slučaju povećanja zahteva korisnika sistema (veći broj korisnika, veći broj transakcija od strane korisnika i sl.).

Druga primena rezultata je za planiranje akcija koje mogu da se sprovedu kao privremena mera, pre nego što je moguće ispraviti greške koje se javljaju u sistemu. Primer za ovo su situacije u kojima softver ne oslobađa resurse koje više ne koristi. Ovakvi problemi se rešavaju instaliranjem nove verzije softvera, ali dok se nova verzija ne razvije, testira i pokrene, uobičajena praksa je da se aplikacija planski periodično restartuje da bi resursi bili oslobođeni.

Tema ove disertacije je razvoj sistema za kontinualno praćenje performansi softvera, kao i razvoj modela za predviđanje performansi softvera. Za implementaciju sistema upotrebljena je JEE tehnologija, ali je sistem razvijen tako da može da se primeni i za praćenje softvera razvijenog

za druge platforme. Linearna regresija je upotrebljena za modelovanje zavisnosti performansi od okruženja u kom se softver izvršava.

Ciljevi disertacije su:

- razvoj i implementacija sistema za kontinualno praćenje rada distribuiranih aplikacija,
- automatizacija procesa utvrđivanja poštovanja ugovora o nivou servisa - (eng. *service level agreement*, skr. SLA),
- modelovanje budućeg ponašanja aplikacije u zavisnosti od parametara okruženja u kom se aplikacija nalazi i na kakvoj platformi se izvršava,
- definisanje procedure za održavanje aplikacije u uslovima kada ona ne funkcioniše potpuno u skladu sa SLA

Sistem za praćenje treba da vrši svoju funkciju uz što manji uticaj na softver koji prati. Da bi to ostvario, sistem treba da se prilagođava na osnovu dobijenih podataka. Praćenje treba da ide u smeru otkrivanja greške, a da se u delovima softvera u kojim ne postoji problem vrši samo površno praćenje, za svaki slučaj, ako se neki problem pojavi. Ovo se odnosi i na praćenje poštovanja SLA: praćenje treba da se uključuje samo tamo gde se utvrdi nepoštovanje SLA.

U cilju modelovanja budućeg ponašanja aplikacije koristi se linearna regresija.

Tekst disertacije se sastoji iz šest poglavlja.

U prvom poglavlju su data uvodna razmatranja iz oblasti praćenja i predikcije rada softverskih aplikacija.

U drugom poglavlju dat je pregled postojećih naučnih radova i rezultata iz ovih oblasti. Postojeći radovi su podeljeni u tri glavne kategorije:

- istraživanja iz oblasti praćenja rada softverskih aplikacija
- definisanje kvaliteta softverskih servisa
- istraživanja iz oblasti predikcije ponašanja softverskih aplikacija

Model sistema za adaptivno praćenje softverskih aplikacija – DProf – dat je u trećem poglavlju. Pošto je DProf sistem zasnovan na Kieker okruženju, u ovom poglavlju je dat opis tog okruženja. Data je i XML shema kojom se definišu očekivani nivoi softverska čiji rad se prati. Formalnu specifikaciju samog DProf sistema čine:

- definicije i specifikacija osnovnih komponenti sistema pomoću UML-a
- osnovne odrednice u procesu prikupljanja podataka o izvršavanju aplikacije
- opis procesa analize prikupljenih podataka i procesa izmene konfiguracije praćenja pomoću UML-a

- opis pripreme sistema za rad

Prikaz upotrebe sistema za praćenje jedne distribuirane Java aplikacije u različitim konfiguracijama dat je u četvrtom poglavlju. Poglavlje sadrži prikaz test konfiguracija na kojima je pokrenuta aplikacija i opis dve konfiguracije sistema za praćenje: jedne koja se koristi za praćenje brzine odziva i druge, koja se koristi za praćenje poštovanja ugovora o nivou servisa. Ovaj drugi primer konfiguracije prikazuje adaptivnost procesa praćenja u cilju otkrivanja uzroka odstupanja. U ovom poglavlju je izvršena analiza uticaja DProf sistema za praćenje na performanse aplikacije koja se prati.

Primena dobijenih rezultata za predikciju ponašanja aplikacije data je u petom poglavlju. Prikazana su dva modela. Prvi model se koristi za predviđanje brzine odziva aplikacije i broja neobrađenih zahteva. Drugi model se koristi za planiranje preventivnog restartovanja aplikacije.

Poslednje poglavlje donosi zaključna razmatranja, analizu naučnog i praktičnog doprinosa disertacije i pravce daljih istraživanja.

Na kraju disertacije dat je spisak korištene bibliografske građe koja je pomenuta u tekstu disertacije.

U tekstu su korišćene sledeće konvencije. Sve skraćenice objašnjene su prvi put kada su pomenute u tekstu tako što je naveden izraz na koji se skraćenica odnosi, a zatim je u zagradi navedena sama skraćenica. Strane reči i izrazi napisani su iskošenim slovima. Programski kod, nazivi klasa, atributa i metoda i XML kod napisani su neproporcionalnim pismom.

Zahvaljujem članovima komisije na pomoći u istraživanju i izradi doktorske disertacije. Posebnu zahvalnost dugujem mentoru prof. dr Milanu Vidakoviću i prof. dr Zori Konjović za nesebičnu podršku tokom rada na disertaciji.

Novi Sad, septembar 2012.

Dušan Okanović

Sadržaj:

1 Uvodna razmatranja

Moderne softverske aplikacije i servisi su sve veći i kompleksniji. Pri njihovom razvoju potrebno je da se ima u vidu da nije dovoljno da softver radi ono za šta je namenjen tj. da ispunjava funkcionalne zahteve. Naime, softver mora da to radi na određen način tj. u skladu sa nefunkcionalnim zahtevima. Stepen zadovoljenja njegovih nefunkcionalnih zahteva predstavlja nivo softverskog servisa [Clifford08]. Primeri nefunkcionalnih zahteva su dostupnost, stabilnost, portabilnost, vreme odziva, otpornost, robusnost, itd. Nefunkcionalni zahtevi predstavljaju deo ugovora između proizvođača softvera i naručioca, zajedno sa funkcionalnim.

Biblioteka IT infrastrukture (eng. *Information Technology Infrastructure Library* – ITIL) [Meyer05] definiše, između ostalog, skupove procedura za upravljanje isporukom servisa (eng. *service delivery*). Jedan deo procedura definiše i upravljanje nivoom servisa (eng. *service level management* – SLM) – trajnu identifikaciju, praćenje i analizu nivoa IT servisa definisanih u okviru ugovora o nivou servisa (eng. *Service Level Agreement* – SLA).

Na kvalitet isporuke servisa, samim tim i na njegov nivo, utiče više faktora. Hardverske tehnike koje se koriste u održavanju kvaliteta servise su dobro poznate i najčešće se svode na upotrebu redudantnog hardvera, rezervnih sistema i sl.

Pojam "starenje softvera" (eng. *software aging*) opisuje situaciju u kojoj softver je dostupan, ali nivo servisa vremenom opada. Odnosi se na nefunkcionalne zahteve ili tačnije, performanse servisa. Pojam je prvi put uveden u upotrebu u [Huang95].Treba napomenuti da se pojam "starenja" koji se ovde koristi, razlikuje od pojma "zastarevanja". Pojam "zastarevanje", definisan u [Parnas96], se odnosi na situaciju u kojoj se vremenom menjaju zahtevi (i funkcionalni i nefunkcionalni), pa softver više ne može da ih ispunjava. Problem "zastarevanja" se se obično rešava prelaskom na neku drugu (noviju) verziju softvera.

Uprkos napretku u razvoju formalnih metoda i metodologija za razvoj softvera i njegovo testiranje, na nivo kvaliteta servisa najviše utiču greške koje se pojavljuju u softveru (eng. *bugs*). Istraživanje [Holzmann07] pokazuje da se u svakom softveru, uz temeljno i kvalitetno testiranje i proveru, u proseku nađe 0.1 defekt na 1000 linija koda. Drugo istraživanje prikazano u [Agarwal04] pokazuje da je ukupna cena nekog servisa 5-10 puta veća od cene plaćene za servis i potreban hardver i softver. Od te, ukupne, sume, od jedne trećine do čak jedne polovine se utroši na

ispravljanje grešaka ili bar na pripremu za pojavu mogućih problema u sistemu.

Postojanje greške u kodu softvera ne mora nužno da vodi do toga da servis momentalno postane nedostupan tj. da prestane da ispunjava zahteve. Česte su situacije u kojima softver vremenom počinje da ima sporiji odziv, da se javljaju zastoji i sl.

Greške koje postoje u programskom kodu servisa mogu da dovedu do toga da se softver nađe u stanju interne greške (eng. *error*). Stanje interne greške znači da softver u toku daljeg rada može da pređe u stanje pada (eng. *failure*). Ako, umesto pada, pređe u drugo stanje greške, govori se o propagaciji greške.

Recimo da imamo program koji radi sa nekakvim objektima. Tokom izvršavanja tog programa program kreira objekte i radi sa njima. Primer greške u kodu može da bude izostanak dealokacije memorije koju zauzima neki objekat. Kada se takav program pokrene, on počinje da kreira objekte i da zauzima memoriju. U slučaju kada neki objekat postane nepotreban, referenca na taj objekat će da nestane, ali memorija koju on zauzima se ne oslobađa. Program je u tom slučaju u stanju interne greške – radi u skladu sa funkcionalnim zahtevima, ali ne radi to na predviđen način. Ako softver radi neko duže vreme, prvo što može da se desi u praksi je da on počne da radi sporije. Obično jer operativni sistem sve više počinje da koristi mehanizam straničenja da bi obezbedio dovoljno radne memorije za program. Dalje izvršavanje softvera može da dovede do pada celog sistema ili bar prekida rada softvera, ako se potroši i sva dostupna virtuelna memorija i tada sistem prelazi u stanje pada.

U [Avižienis04] prikazana je shema za klasifikaciju grešaka po nekoliko kriterijuma. Prema ovoj shemi softverske greške (*bugs*) predstavljaju interne, nenamerne trajne greške u softveru izazvane ljudskim faktorom. Softverske greške dalje mogu da se podele na "čvrste" i "nedokučive" greške. "Čvrste" greške su one čija se aktivacija i propagacija može reprodukovati. "Nedokučive" (nazvane i "meke") greške su one koje se ne ponavljaju u sistemu uvek na isti način. Ova podela grešaka je subjektivna, jer greška može različitim korisnicima biti različito "nedokučiva", u zavisnosti od toga koliko oni poznaju posmatrani softver.

Objektivnija podela grešaka je data u [Grottke07]. Ponavljanje određenih koraka ne mora da vodi novoj pojavi greške – replikaciji greške (tj. stanja greške). Na prelazak u stanje greške mogu da utiču i drugi faktori računarskog sistema koji nisu deo aplikacije i označavaju se zajedničkim terminom "interno okruženje sistema" (eng. *system internal environment*). U "interno okruženje sistema" spadaju svi elementi računarskog sistema na kom se softver izvršava, a koji nisu deo posmatranog softvera – npr.

hardver na kom se posmatrani softver izvršava, operativni sistem, druge aplikacije koje se izvršavaju na računarskom sistemu, a nisu deo posmatranog softvera (a sa kojima posmatrani softver može da interaguje), itd. Definicija se odnosi samo na ono što se nalazi u granicama računara, tj. u "interno okruženje" ne spadaju npr. korisnici, administratori, itd.

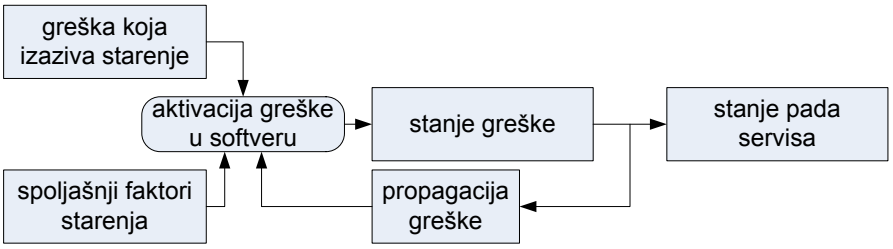
Kombinacija grešaka u softveru i uticaja ovih faktora može da izazove haotično ponašanje i nedeterminističke prelaze u stanje pada. Zbog toga, ovakve greške se obično nazivaju Mandel-greške, kao aluzija na Benoa Mandelbrota, francuskog matematičara i vodećeg istraživača na polju fraktalne geometrije. Karakteristično za Mandel-greške je to što se gotovo nikad ili vrlo teško, otkrivaju u toku testiranja, jer sistem na kom se vrši testiranje ne mora da predstavlja isto unutrašnje okruženje aplikacije kao sistem na kom će aplikacija biti pokrenuta.

Greške opisane terminom Hajzen-greške se često razmatraju posebno. One su definisane kao greške koje menjaju svoje ponašanje kada se pokuša izolovanje. Pošto alati koji se koriste za traženje grešaka ulaze u sastav "internog okruženja aplikacije", ove greške spadaju u Mandel-greške.

Pojam suprotan pojmu Mandel-greška je Bor-greška, nazvan po Nilsu Boru, danskom fizičaru. Bor-greške se lako izoluju i uvek se manifestuju na isti način.

Starenje softvera je direktna posledica Mandel-grešaka: usled izvršavanja softvera u nekom okruženju i grešaka koje postoje u tom softveru, softver više ne zadovoljava SLA. Softver praktično dolazi u stanje u kom je npr. odziv sporiji u odnosu na odziv na početku, zauzima više memorije, i sl. Greške koje izazivaju starenje softvera uvek spadaju u Mandel-greške jer vreme koje protekne od pojave stanja greške do stanja pada je dovoljno veliko da se akumulira dovoljno grešaka i može da zavisi od internog okruženja aplikacije.

Mehanizam koji opisuje korelaciju između grešaka u kodu, okruženja u kom se softver izvršava i starenja softvera prikazan je na slici 1.1.



Slika 1.1 Korelacija između greške u softveru i spoljašnjih faktora i kako to utiče na prekid rada softvera

Kombinacija grešaka u kodu koje mogu da izazovu starenje softvera i spoljašnjih faktora starenja dovodi softver u stanje greške. Dalje ponašanje sistema zavisi od same prirode greške. Moguća su dva scenarija: da softverski servis ode u stanje pada ili da se greška dalje propagira, da izaziva nove greške i da se vrši akumulacija grešaka. Akumulacija izaziva efekat starenja – postepeno odstupanje softvera od zadatog ponašanja. Greške koje odmah dovode do pada obično se lako otkrivaju već u fazi testiranja softvera.

Da bi se proverilo da li softver zadovoljava postavljene zahteve, tj. da li ispunjava zadate nivoe servisa, kao i da li radi u predviđenim okvirima (zauzeća memorije, procesora, itd) postoji potreba da se rad softvera stalno prati i analizira. Stalno praćenje i analiza mogu da preduprede pojavu ozbiljnih zastoja u sistemu.

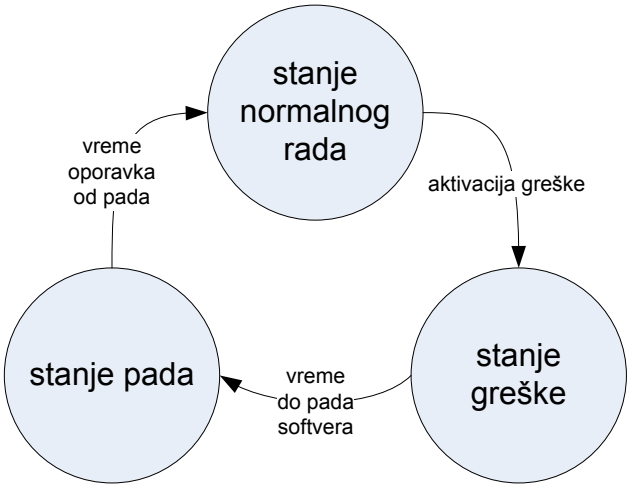
Podaci dobijeni stalnim praćenjem mogu da nam ukažu na postojanje problema u radu softvera – npr. da ukažu na akumulaciju greške – i ostavljaju nam dovoljno vremena da reagujemo na njega. Ako softver ne pratimo, nego samo čekamo da se pojavi greška ili da softver pređe u stanje pada i onda ispravljamo grešku, onda imamo reaktivan pristup upravljanju softverom. Ako se dobijeni podaci iskoriste za predviđanje ponašanja softvera i predupređivanje negativnih događaja, onda imamo proaktivan pristup održavanju softvera.

U toku procesa trajnog praćenja mogu da se pronađu prvo indikatori da nešto nije u redu, a tek onda i tačan uzrok. Identifikacija problema, tj. pronalaženje tačne lokacije greške u kodu softvera ili faktora starenja, koji utiču na to da aplikacija pređe u stanje greške, može da bude dugotrajan posao. Sledeći korak je uklanjanje dela koda softvera u kom se nalazi greška ili rešavanje pitanja faktora u okruženju aplikacije koji izazivaju starenje softvera (npr. odstupanje od očekivanih vrednosti definisanih u SLA). Nakon toga sledi novi krug testiranja i pokretanja nove verzije aplikacije.

Pošto svi ovi koraci zahtevaju vreme, postavlja se pitanje šta da se radi sa softverom u međuvremenu. Recimo da imamo softver koji opslužuje veliki broj klijenata i čiji prekid rada može skupo da košta pružaoca usluge. Tokom procesa praćenja utvrđeno je da postoji curenje memorije. Dalje praćenje utvrdiće da postoji greška u jednom delu softvera, ali do ispravke greške mora da se nađe način da se predupredi prelazak u stanje pada. Jedan od načina je da se softver periodično zaustavi i ponovo pokrene. Ovo je nešto što se često koristi u praksi, najčešće od strane običnih korisnika, ali se pokazuje kao korisno. U literaturi se pojavljuje tek od skoro i poznato je pod nazivom podmlađivanje softvera (eng. *software rejuvenation*) [Siewiorek92].

Podmlađivanje softvera podrazumeva postupak u kom se periodično vrši zaustavljanje i ponovno pokretanje aplikacije. Na taj način aplikacija se dovodi u početno stanje i oslobađaju se blokirani resursi. Npr. oslobađa se zauzeta memorija, konekcije prema bazama podataka, datotekama.

Recimo da se rad softvera može prikazati sledećim dijagramom stanja [Grottke07].



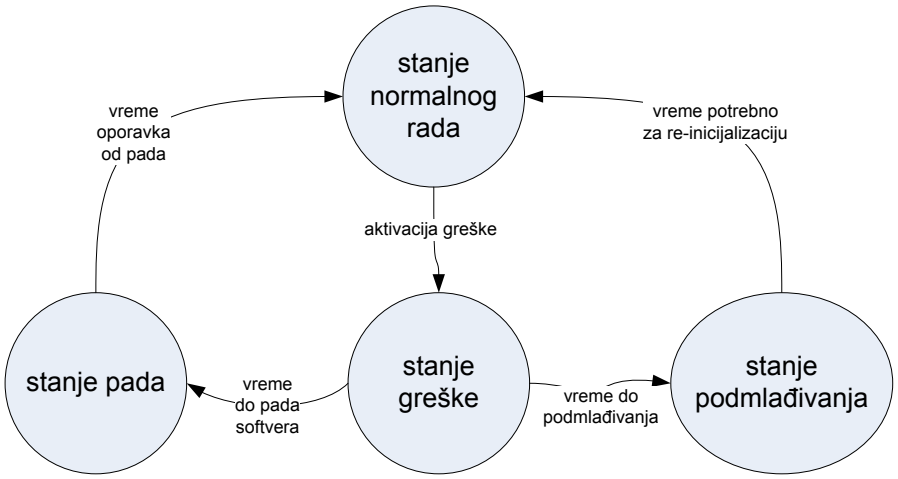
Slika 1.2 Dijagram stanja rada softvera u kom postoji greška

Softver iz stanja u kom normalno radi, prelazi u stanje greške, a zatim u stanje pada. Kada se primeti pad, vrše se koraci potrebni za oporavak i ponovno pokretanje aplikacije. Kako je uobičajena situacija da se stanje pada ne primeti odmah nego posle određenog vremena, sve ovo može da izazove probleme sa korisnicima servisa.

Stalnim praćenjem rada softvera moguće je detektovati kada je softver prešao u stanje greške. Ako primenimo podmlađivanje softvera, onda se u sistem uvodi novo stanje – stanje podmlađivanja.

U toku stanja podmlađivanja može da se izvrši zaustavljanje softvera i ponovno pokretanje koje će da vrati softver u inicijalno stanje, tako da on nastavlja normalno da radi. Podmlađivanje treba da se vrši pre nego što softver pređe u stanje pada. Pošto stanje greške može da traje izvesno vreme, nema smisla podmlađivati aplikaciju svaki put kad pređe u stanje greške, nego to treba da se radi prema planu.

Upravo kod pravljenja plana podmlađivanja do izražaja dolazi stalno praćenje aplikacije. Pomoću alata za stalno praćenje moguće je da se prati potrošnja resursa i na osnovu dobijenih podataka da se detektuje kada je sistem ušao u stanje greške i kada može da dođe do stanja pada.



Slika 1.3 Dijagram stanja rada softvera sa uključenim stanjem podmlađivanja softvera

Osim podmlađivanja, proaktivno delovanje u održavanju softvera može da se primeni u situaciji kada softver radi normalno (recimo da nema prelaska u stanje greške ili je uticaj grešaka minimizovan podmlađivanje). Podaci dobijeni stalnim praćenjem mogu da se iskoriste za ravnomerno raspoređivanje opterećenja između više servera. Ako jedan od servera radi sporije od ostalih, to može da se utvrdi stalnim praćenjem, pa raspoređivač može taj server opterećuje manje.

Stalno praćenje može da se iskoristi i u situaciji kada zbog povećanog broja klijenata aplikacija ne može pravovremeno odgovori na sve zahteve. Rezultati praćenja mogu da nam pokažu za koji broj klijenata dostupni resursi daju zadovoljavajući odziv, a kada treba da se uključe dodatni resursi (npr. da se dodaju novi serveri, memorija, hard-diskovi, ...).

U postupku analize aplikacija i servisa postoje dva osnovna pristupa: statička i dinamička analiza.

Statička analiza programa se vrši bez pokretanja samog programa. U većini slučajeva analiza se vrši na izvornom kodu programa ili, ređe, na nekom obliku objektnog koda (npr. Java *bytecode*, *common intermediate language* za .NET). Termin analiza se ovde obično odnosi na postupak automatskog prikupljanja podataka, dok se analiza koju sprovodi čovek obično zove razumevanje (eng. *comprehension*).

Ciljevi statičke analize mogu da budu od traženja grešaka u kodu do formalne provere da li program radi ono šta je predviđen da radi. Za

izvršavanje ovih provera razvijene su brojne formalne metode. Neke od njih su:

- provera modela (eng. *model checking*) [Emerson81] – automatska provera da li je model u skladu sa zadatom specifikacijom. Obično se proverava da li softver može da uđe u mrtvu petlju ili da li softver sadrži kod koji može da izazove pad,
- analiza toka podataka (eng. *data flow analysis*) [Hecht77] – analiza tokom koje se posmatra skup promenljivih u programu i traži povezanost između dobijenih vrednosti u cilju optimizacije izvršnog koda od strane kompajlera,
- apstraktna interpretacija (eng. *abstract interpretation*) [Cousot77] – analiza tokom koje se vrši procenjivanje funkcionalnosti programa. Rezultat je matematička karakterizacija mogućeg ponašanja programa. Tokom analize pravi vodi se računa o balansu između preciznosti dobijene karakterizacije i njene kompleksnosti,
- Hoarova logika (*Sir Charles Antony Richard Hoare*) [Hoare69] – predstavlja formalan sistem sa skupom logičkih pravila koja se koriste za proveru korektnosti programa. Moguće je proveravati parcijalnu i totalnu korektnost.

Statičku analizu ima smisla primenjivati u toku razvoja softvera. Navedene tehnike su obično ugrađene u alate za razvoj, npr. postoje alati za proveru korektnosti modela, analiza toka podataka se koristi u kompajlerima, dok se apstraktna interpretacija koristi u dibagerima (eng. *debugger*) i kompajlerima. Kompajleri koriste apstraktnu interpretaciju da provere da li je moguće da se izvrši neka transformacija u cilju optimizacije izvršnog koda. U procesu dibagovanja, apstraktna interpretacija se korsti za proveru programa u odnosu na klase grešaka koje mogu da se jave.

Nedostatak statičke analize je u tome što nije moguće dobiti sliku o ponašanju softvera kada se on pokrene u realnom okruženju.

Dinamička analiza programa je analiza softvera koja se vrši tokom njegovog izvršavanja na stvarnom ili virtualnom procesoru. Neophodno je da se program izvrši dovoljan broj puta da bi se dobili reprezentativni rezultati. U dinamičku analizu spadaju dibageri, profajleri (eng. *profiler*), *memory leak* i *memory corruption* detektori, kao i drugi alati za testiranje.

Profajliranje je proces automatskog prikupljanja podataka o izvršavanju programa i njihovog prikazivanja u nekom obliku. U stručnoj literaturi ovaj proces se naziva i profajliranje programa (eng. *program profiling*), profajliranje softvera (eng. *software profiling*) ili samo profajliranje. Alat koji se koristi u postupku profajliranja naziva se profajler. Pored profajliranja, ravnopravno se koristi i pojam stalnog (kontinuiranog) praćenja. Ako je potrebno da se napravi razlika, onda se,

najčešće, pojam „profajliranje“ koristi za aktivnosti u toku razvoja aplikacije, a „kontinuirano praćenje“ kada aplikacija uđe u upotrebu.

Kao i svaka dinamička analiza, profajliranje predstavlja ispitivanje ponašanja programa pomoću informacija, prikupljenih dok se program izvršava. Da bi analiza imala stvarni efekat, ciljni program mora da bude izvršen dovoljan broj puta, sa odgovarajućim ulaznim podacima.

Uobičajeni cilj ove analize je optimizacija programa u smislu brzine izvršavanja, alokacije memorije i drugih sistemskih resursa. Osim ovoga, profajliranje se koristi za optimizaciju kompajlera, poređenje performansi programa sa performansama zadatim specifikacijom, kao i za predikciju ponašanja softvera u zavisnosti od opterećenja (eng. *profiling based prediction*).

Rezultati dobijeni profajliranjem mogu da imaju različitu namenu [Srivastava94]. Projektanti hardvera koriste rezultate da provere kako se programi izvršavaju na novim arhitekturama. Programeri pomoću njih analiziraju svoje programe da bi utvrdili da li se njihov softver ponaša u skladu sa očekivanim performansama (koje mogu da budu i deo specifikacije zahteva) i koji deo koda da se optimizuje da se postignu zadovoljavajuće performanse (eng. *performance engineering* – skup aktivnosti, tehnika, alata, itd. koji se koriste u svakoj fazi životnog ciklusa softvera, a koji treba da osiguraju da će softver biti projektovan, implementiran i podržan tako da podrži nefunkcionalne zahteve kao što su performanse [Pooley2000]). U dizajniranju kompajlera, rezultati profajlera koriste se za proveru optimizacionih tehnika [Gatlin05].

U razvoju alata za praćenje performansi koristi se niz metoda.

Kod metoda zasnovanih na registrovanju događaja (eng. *event-based*) profajleri se realizuju kao agenti koji se povezuju na platformu na kojoj se izvršava program i beleži događaje od značaja. Ovaj metod je zastupljen kod Java i .NET platforme, gde se profajler preko odgovarajućeg API-ja povezuje na virtuelnu mašinu. U slučaju Java platforme koristi se JVM TI (eng. *Java Virtual Machine Tools Interface*). U slučaju .NET platforme agent se povezuje kao COM (eng. *Component Object Model*) komponenta na CLR (eng. *Common Language Runtime*). U oba slučaja virtuelna mašina obezbeđuje interfejs za pristup; mana ove metode je što se ovakvi profajleri pišu u nekom drugom jeziku, a ne u jeziku platforme (recimo C/C++ za JVM TI).

Profajleri koji su zasnovani na uzorkovanju (eng. *sampling*) periodično, pomoću prekida na nivou operativnog sistema, očitavaju stanje programskog brojača; rezultati koji se dobijaju ovom metodom nisu potpuno precizni, jer može da se desi da se neki događaji propuste. Prednost ovog pristupa je što posmatrani program može da radi (gotovo) punom

brzinom i dobija se najmanja greška merenja. Nedostatak pristupa je taj što su obično zasnovane na tehnikama koje su veoma zavisne od platforme na kojoj se izvršavaju jer se koriste prekidi, sistemski sat, raspoređivač (eng. *scheduler*), programski brojač ili softverske kuke na platformi [Inoue09].

Iako instrumentacija – ubacivanje dodatnog koda u program – može da izazove ozbiljne poremećaje u performansama, uz dobro praćenje i kontrolu može da obezbedi dobre rezultate. Greška merenja može i da se izmeri, pa da se isključi iz rezultata. Nedostatak pristupa je taj što se programski kod "prlja" kodom koji se koristi za praćenje, što otežava kasnije održavanje softvera.

Posebna vrsta instrumentacije je instrumentacija u okviru interpretera. Pošto interpreteri imaju punu kontrolu nad izvršavanjem programa, oni mogu da izvrše instrumentaciju programa na osnovu zadatih pravila i da daju kvalitetne rezultate.

Hipervizori su programi koji omogućuju izvršavanje više operativnih sistema na jednom računaru, pa mogu da vrše praćenje rada programa jer imaju pun pristup resursima operativnog sistema.

Za profajliranje mogu da se koriste i simulatori skupa instrukcija – programi koji simuliraju rad procesora. Ovi simulatori su najčešće implementirani kao *debugger*-i i koriste se za ispitivanje programa. Problem sa ovim simulatorima je taj što ne daju stvarnu sliku o izvršavanju programa u realnom sistemu.

Nakon primene nekog od ovih pristupa, dobijaju se rezultati u različitom obliku:

- statistički obrađeni zabeleženi događaji – podaci dobijeni merenjem se obično prikazuju uz kod čije performanse mere, za svaki deo koda posebno,
- niz zabeleženih događaja (eng. *trace*) – ako je potrebno događaje posmatrati u određenom kontekstu, onda se posmatra ceo niz događaja,
- kontinualan tok rezultata – ako se u profajliranju koristi interakcija sa hipervizorom, u okviru kog se program izvršava, onda rezultati mogu da se kontinuirano prikazuju, u toku izvršavanja programa.

Statistički obrađeni događaji su pogodni kod programa koji se sekvencijalno izvršavaju, ali ako je potrebno pratiti rad programa u kojima postoji paralelizam, onda je potrebno rezultate prikazati u određenom kontekstu. Profajleri koji rade sa celim nizom događaja mogu da daju i graf poziva (eng. *call graph*), a ne samo dobijene vrednosti merenja, tako da se rezultati posmatraju u odgovarajućem kontekstu. Svaki čvor u ovom grafu predstavlja jednu proceduru (funkciju, metodu), a svaki prelaz između dva čvora predstavlja poziv jedne procedure iz druge.

Merenje performansi softvera može da unese izmene u njegovom ponašanju – npr. usporenje u izvršavanju, povećano zauzeće memorije, povećan mrežni saobraćaj, itd. Greška merenja (eng. *performance overhead*) koja se javlja u ovom slučaju mora da se predvidi i da se njen uticaj smanji ili eliminiše.

Stvari se dodatno komplikuju u slučaju distribuiranih aplikacija. Distribuirane aplikacije se izvršavaju na heterogenim platformama, počevši od različitog hardvera, operativnih sistema, podrazumevaju upotrebu različitih komunikacionih mreža, itd. Distribuirani sistemi mogu da se sastoje od velikog broja čvorova koji međusobno komuniciraju, pa je profajliranje od velikog značaja za praćenje performansi i pouzdanosti sistema. Praćenjem rada mogu da se utvrde odstupanja od predviđenih performansi, pronađu uska grla i greške, još u toku razvoja, ali praćenje rada ovih aplikacija mora stalno da se vrši. Vrlo često se dešava da performanse softvera variraju tokom vremena, pa je neophodno da se utvrde i uklone uzroci ovoga.

U ovom radu biće prikazan alat, pomoću kog može da se vrši kontinuirano praćenje distribuiranih aplikacija, uz malu grešku merenja. Može da se vrši provera usklađenosti sa SLA ili samo da se vrši merenje (u određenoj metrici). U radu je prikazana specifikacija i neki detalji implementacije sistema, kao i XML shema dokumenta za definisanje nivoa servisa. Upotreba sistema je prikazana na praćenju rada jedne Java EE aplikacije.

Neki delovi ove teze objavljeni su na konferencijama i stručnim časopisima. Praćenje rada distribuiranih aplikacija pomoću Kieker okruženja je prikazano u [Okanovic10]. Arhitektura DProf sistema prikazana je u [Okanovic11a], a analiza rezultata dobijenih prikupljanjem u [Okanovic11b]. DProfSLA shema je prikazana u [Okanovic11c], a praćenje poštovanja ugovora napravljenog po ovoj shemi u [Okanovic12a] i [Okanovic12b]. Predikcija performansi softvera na osnovu prikupljenih rezultata prikazana je u [Okanovic12c] i [Okanovic12d].

2 Pregled postojećih istraživanja

Alati, koji se koriste u postupku praćenja, obično mere učestanost i trajanje poziva metoda i alokaciju resursa (zauzetost RAM memorije, pristup masovnoj memoriji, mrežni saobraćaj, ...) u toku njihovog izvršavanja na stvarnom računaru.

Prvi alati za analizu performansi računarskih programa postojali su na IBM/360 i IBM/370 računarima (u ranim 1970-im) i obično su u pravilnim vremenskim razmacima sakupljali podatke o stanju programa na osnovu registara. 1974. pojavili su se prvi simulatori koda koji su omogućavali beleženje događaja tokom izvršavanja simulacije. Na osnovu zabeleženih podataka, vrši se analiza performansi.

Na Unix sistemima, alat *prof* iz 1979. beležio je izvršavanje svake funkcije i njeno trajanje [Prof79]. 1982. pojavio se *gprof* [Graham82]0, koji je proširio *prof* analizom grafa poziva. *gprof* se danas koristi u okviru *GNU Binary Utils* [GNUBinutils] komponente GNU projekta [GNUOS] i može da se koristi za profajliranje programa napisanih u bilo kom programskom jeziku podržanom od strane GNU kolekcije kompajlera [GNUGCC] (eng. GNU Compiler Collection).

Početkom 90-ih godina dvadesetog veka pojavilo se nekoliko alata za profajliranje u okviru kojih se vrši neka vrsta instrumentacije. Jedan deo ovih alata vrši brojanje osnovnih blokova (eng. *basic block counting*). Ovaj pristup se zasniva na posmatranju osnovnih blokova i njihovom trajanju. Pod osnovnim blokom ovde se smatra kontinualni linearni segment koda, bez grananja. Linije koda u okviru osnovnog bloka se izvršavaju jednak broj puta u toku izvršavanja programa. Analiza osnovnih blokova (prepoznavanje osnovnih blokova u programskom kodu) se vrši odmah nakon kompajliranja i za svaki osnovni blok se definiše brojač. Ovakvo profajliranje obično proizvodi datoteku koja je kopija izvornog koda, gde je, uz svaki osnovni blok, dodat broj koji predstavlja broj izvršavanja tog osnovnog bloka. Jedan od prvih alata koji koriste ovaj pristup je *pixie* [Pixie89], koji je razvijen za instrumentaciju programa koji se izvršavaju na MIPS procesorima. U [Larus94] predstavljeni su predlozi pomoću kojih se proces profajliranja unapređuje: predlaže se način da se analiza vrši nakon kompajliranja i daju se preporuke za izmenu formata izvršnih datoteka da bi se profajliranje učinilo efikasnijim. U radu je prikazan i alat *qpt*. Oba alata su namenjena isključivo za MIPS procesore i programski jezik C. Jedino profajliranje koje mogu da vrše je brojanje osnovnih blokova i svaka izmena, bilo da želimo da dobijemo neku drugu vrstu informacije (količina memorije, mrežni protok, ...) ili da dobijemo više ili manje informacija od

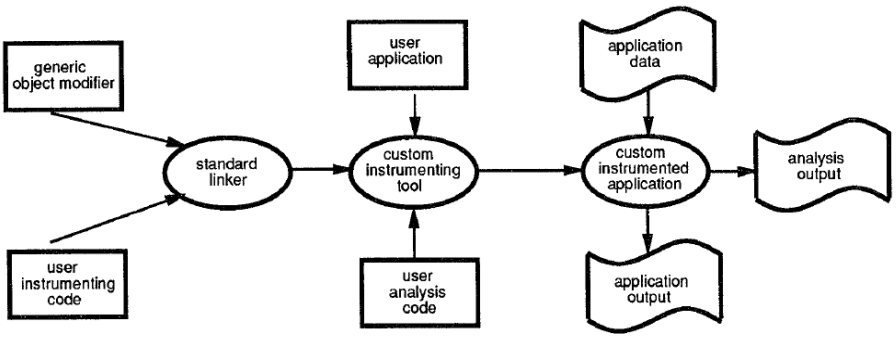
alata, zahteva izmenu u alatu i može da predstavlja ozbiljan dodatni napor [Srivastava94]. Uz to, *qpt* može ozbiljno da naruši performanse posmatrane aplikacije pošto zauzima tri procesorska registra za svoje potrebe, a aplikaciju preusmerava tako da ona koristi tri lokacije u radnoj memoriji.

ATOM [Srivastava04] predstavlja sistem za izradu alata za profajliranje. Na osnovu ovog sistema mogu da se prave alati koji prevazilaze nedostatke alata [Pixie89] i [Larus94]. Sistem je razvijan za MIPS procesore, ali se kasnije prešlo na DEC Alpha procesore. On predstavlja jedan od prvih alata u kojima se koristi instrumentacija za sakupljanje različitih vrsta podataka uz mogućnost izbora dela koda koji se posmatra.

Pomoću ATOM-a mogu da se kreiraju alati koji vrše različite vrste profajliranja [Eustace95] i dozvoljava selektivnu instrumentaciju. U projektovanju ATOM sistema, pošlo se od ideje da iako se sakupljaju različiti tipovi informacija svi se sakupljaju na isti način – instrumentacijom programa na određenim lokacijama u programskom kodu. Bez obzira šta se posmatra, uvek postoji ista infrastruktura koja povezuje instrumentaciju ubačenu u aplikaciju sa rutinama koje sakupljaju i analiziraju podatke.

Kreiranje zajedničke infrastrukture za različite tipove podataka koji se sakupljaju je moguće jer ATOM posmatra program kao linearnu kolekciju procedura, procedure kao kolekciju osnovnih blokova, a blokove kao kolekciju instrukcija. Korisnik može sam da bira koji delovi koda treba da se instrumentiraju i koji podaci da se sakupljaju.

Rad ATOM sistema na instrumentaciji aplikacije se sastoji nekoliko koraka. Ceo proces može da se predstavi dijagramom datim na slici 2.1.



Slika 2.1 Dijagram upotrebe ATOM procesa (preuzeto iz [Srivastava94])

Korisnik treba sistemu da prosledi objektni modul same aplikacije koja se instrumentira, datoteku sa pravilima za instrumentaciju (u kojoj se

navode lokacije u programskom kodu aplikacije gde treba da se umeće instrumentacija) i datoteku sa rutinama za analizu (procedure koje se pozivaju od strane umetnute instrumentacije). ATOM kombinuje instrumentacionu infrastrukturu (koja je zajednička i koristi se u svim profajliranjima) sa korisničkim instrumentacionim rutinama u alat za instrumentaciju zadate aplikacije. Alat dobijen u prethodnom koraku se kombinuje sa aplikacijom i dobija se izvršna datoteka aplikacije sa ubačenim instrumentacionim kodom. Da bi se održalo čisto izvršavanje aplikacije, bez ometanja od strane ATOM sistema, instrumentacioni kod i rutine za analizu ne koriste ništa od procedura i biblioteka posmatrane aplikacije, nego imaju svoje.

Izvršna datoteka instrumentirane aplikacije se normalno pokreće, prima ulazne podatke, a kao rezultat, osim svog redovnog izlaza, daje i izlaz koji se koristi za analizu.

Korisničke procedure za analizu prikupljenih podataka se, zahvaljujući ATOM sistemu, izvršavaju u istom adresnom prostoru kao i program čiji se rad prati. Na taj način, podaci se prosleđuju običnim pozivom procedure za obradu, umesto da se snimaju na disk ili šalju putem međuprocesne komunikacije, što bi usporilo ceo sistem i povećalo grešku merenja.

Na tržištu potoji dosta komercijalnih alata koji se koriste za praćenje performansi softvera, ali detalji implementacije uglavnom nisu dostupni. DynaTrace [dynaTrace] koristi svoju PurePath tehnologiju koja beleži vreme i kontekst svake transakcije. Postoje implementacije za Java i .NET okruženja. JXInsight [JXInsight] se koristi za Java EE okruženje. Omogućuje analizu performansi i lokalizaciju problema. IBM's Tivoli Management Framework [JXInsight] je platforma za upravljanje softverskim sistemima. Razvijen je pomoću CORBA standarda. IBM Tivoli Monitoring, koji koristi ovu platformu, je skup alata koji se koriste za prikupljanje podataka o performansama u različitim okoruženjima - JavaEE, .NET, performanse mreže (DNS, DHCP) i dr. AppDynamics omogućava praćenje sa malim uticajem na sistem, ako i automatsku lokalizaciju problema. Postoje i alati za praćenje performansi namenjeni za specifična okruženja - Nagios [Nagios] za infrastrukturu, CA Unicenter [CAUnicenter] za praćenje i upravljanje infrastrukturom i performansama aplikacije i HP Insight [HPInsight] za praćenje i pronalaženje problema na HP serverima.

2.1 Praćenje aplikacija u programskom jeziku Java

Većina alata za praćenje programa pisanih za Java okruženje je bazirana na JVM PI [JVMPI] i JVMTI [JVMTI] specifikacijama.

JVM Profiling Interface (JVM PI) predstavlja interfejs između JVM i agenta za profajliranje. Kroz ovaj interfejs, virtuelna mašina obaveštava agenta o različitim događajima (alokacija objekata, pokretanje programskih niti, i sl). S druge strane, agent, kroz ovaj interfejs, šalje upravljačke signale. Moguće je sakupljati različite podatke u okviru JVM (zauzetost procesora, zauzetost memorije, nepotrebno zadržavanje objekata, i sl). Za jednu virtuelnu mašinu moguće je pokrenuti samo jednog agenta.

JVM PI je razvijen za verziju 1.1 Java okruženja. Osnovni nedostatak interfejsa je taj što njegova upotreba ima ozbiljan uticaj na performanse same virtualne mašine. Prebacivanje na kasnije verzije Java okruženja (pogotovo 1.3) dovelo je do pojave nestabilnosti u radu [OHair04].

Java Platform Debugging Architecture (JPDA) [JPDA] predstavlja skup specifikacija pomoću kojih se vrši ispitivanje Java programa i traženje grešaka (eng. *debug*). U sastav arhitekture ulaze *Java Debugger Interface* (JDI), *JVM Debug Interface* (JVM DI), *Java Debug Wire Protocol* (JDWP) i, već pomenuti, *JVM Tools Interface* (JVM TI).

JVM TI je razvijen u okviru JSR163 [JSR163]. Nalazi se na najnižem nivou JPDA i predstavlja specifikaciju koja omogućava programu da prati izvršavanje aplikacija u JVM i vrši kontrolu nad njima. API definisan u JVMTI obezbeđuje razvojnim alatima (kao što su profajleri i *debugger*-i) pristup stanju JVM.

Biblioteke napisane u C/C++ se učitavaju u toku inicijalizacije JVM. Biblioteke pristupaju JVM preko JVM TI i JNI funkcija i mogu da registruju JVM TI događaje preko event-handler funkcija, kad se dogodi neki događaj u JVM.

Profajliranje pomoću JVM TI se vrši *byte-code* instrumentacijom, a moguće je kreiranje *event-based* profajlera i *sampling* profajlera. JVM TI omogućuje tri vrste *byte-code* instrumentacije:

- statička instrumentacija – klasa se instrumentira pre učitavanja, tako što se napravi njen duplikat koji se instrumentira
- instrumentacija pri učitavanju (eng. *load-time*) – kada se klasa učita u JVM ona se šalje agentu da izvrši instrumentaciju
- dinamička instrumentacija – klase koje su već učitane se instrumentiraju po potrebi; redefinisanje instrumentacije i vraćanje u početno stanje moguće je vršiti proizvoljan broj puta

JVM TI je ušao u široku upotrebu zbog brzine i koristi se uglavnom za razvoj modula za profajliranje u okviru nekih okruženja za razvoj, kao

što je HPROF [HPROF] koji se distribuira uz IBM Java SDK [IBMDK]. Ipak, upotreba JVM TI API-ja ima dva osnovna nedostatka: neophodno je poznavanje C/C++ programskog jezika (da bi se kreirali agenti i vršilo profajliranje) i usporenje i potencijalno velika greška merenja koja nastaje usled činjenice da se komunikacija sa agentom tretira kao *native* poziv.

Jedan od primera upotrebe JVM TI je dat u DYPER [Reiss08]. U radu je predstavljen alat koji vrši profajliranje uzorkovanjem uz garantovano maksimalno kašnjenje aplikacije usled merenja. Alat može da se koristi za praćenje zauzetosti procesora, zauzetosti memorije, I/O komunikacije, alokacije memorije na *heap*-u i ponašanja programskih niti. Za svaku vrstu podataka koji se skuplja postoji poseban proflet – agent koji vrši sakupljanje podataka (uzima u obzir zadati *overhead*) i pakovanje podataka u XML format.

Rezultati prikazani u radu pokazuju da sistem ne utiče previše na performanse same aplikacije, ali i da program, koji se prati tokom dužeg perioda vremena, naglo prekida rad. Zbog ovog problema, sistem nije javno dostupan.

U NetBeans IDE [NetBeans] je uključen JFluid profajler [Dimitriev03]. JFluid predstavlja profajler koji koristi instrumentaciju i razvijen je uz upotrebu JVM TI API-ja. U razvoju ovog alata pošlo se od ideje da se *performance overhead*, koji nastaje usled instrumentacije, eliminiše uključivanjem instrumentacije samo po potrebi. Na ovaj način se smanjuje velika količina informacija koje se dobijaju profajliranjem. Uključivanje i isključivanje instrumentacije je moguće i u toku rada aplikacije.

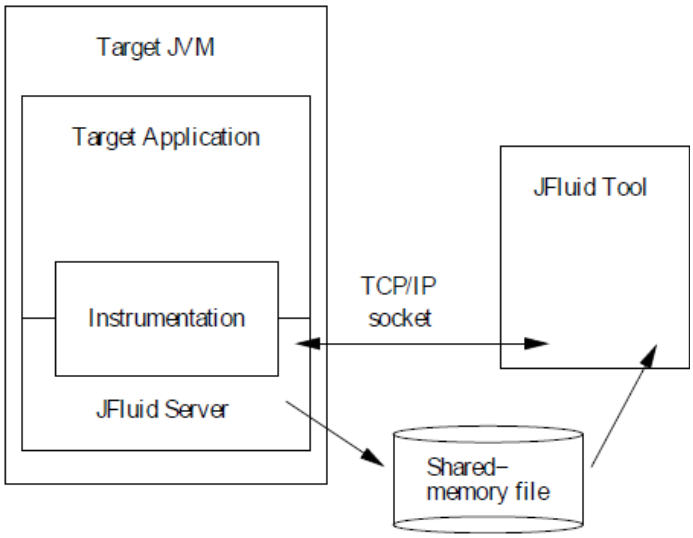
Arhitektura JFluid alata data je na slici 2.2.

JFluid se sastoji od serverske komponente („JFluid Server“) i JFluid alata („JFluid Tool“). Serverska komponenta se izvršava u okviru JVM, zajedno sa posmatranom aplikacijom. JFluid server je napisan delimično u C/C++ programskom jeziku (zbog komunikacije sa JVM TI API-jem), a delimično u programskom jeziku Java (da bi se izbeglo usporenje koje nastaje usled izvršavanja *native* koda napisanog u C/C++-u). Pomoću instrumentacije vrši se prikupljanje podataka.

JFluid alat je obična GUI aplikacija napisana u programskom jeziku Java. U toku izvršavanja aplikacije alat prikuplja i obrađuje dobijene podatke i prikazuje ih korisniku.

Komunikacija između servera i alata se odvija preko TCP/IP protokola i memorijske datoteke. Kroz TCP/IP protokol se vrši prenos upravljanja (od strane alata), kao i nekih manjih količina podataka (odgovori na upravljanje, ako ih ima, i sl). Velike količine podataka koje nastaju u procesu profajliranja prenose se preko datoteke. Ni u jednom od

ova dva slučaja ne koristi se mehanizam serijalizacije jer dodatno usporava rad i pravi *overhead*.



Slika 2.2 Arhitektura JFluid alata (preuzeto iz [Dimitriev03])

JFluid koristi mehanizam dinamičke instrumentacije *byte*-koda (eng. *dynamic bytecode instrumentation*). Pomoću ovog mehanizma moguće je, po potrebi, dodavati u klase i izbacivati *byte*-kod koji vrši instrumentaciju. Iako JVM TI podržava izmenu klasa koje su učitane u JVM, funkcionalnost može da bude problematična ako treba da se instrumentira veliki broj klasa – proces je spor jer se pristupa svakoj pojedinačnoj klasi i može da napravi zagušenje na mreži zbog prenosa velike količine podataka. Rešenje za ovaj problem autori su našli u optimizovanom API-ju koji, umesto redefinisanja cele klase dozvoljava redefinisanje pojedinačnih metoda. Zamenu metode čine dve faze:

- parsiranje *byte*-koda metode i kreiranje nove metode; kreiraju se novi objekti koji predstavljaju nove metode; nove metode JVM proverava standardnim mehanizmom koji se koristi pri učitavanju klasa; koristi se reprezentacija metoda pomoću klase Method (koja je deo Java specifikacije) tako da su metode neke klase predstavljene kao objekti vezani za klase, tj. one su atributi objekata klase Class
- aktivacija instrumentiranog koda – kada se izvrši kreiranje svih novih metoda koje se instrumentiraju i njihova provera, zaustavljaju

se sve programske niti (u stabilnom stanju – eng. *safe point*) i prevezuju reference na objekte

Kada se završi drugi korak, programske niti nastavljaju i sledeći poziv ide instrumentiranoj verziji metode. Važno je napomenuti da se, ako je JVM izvršila određene optimizacije u izvršavanju neke metode koja je instrumentirana, one eliminišu upotrebom mehanizma de-optimizacije, koji je deo HotSpot JVM [HotSpot]. Ovo može da degradira performanse, ali je neophodno da bi opisano redefinisanje metoda moglo da radi.

Postoje dve vrste podataka koje JFluid može da prikuplja i prema formatu podataka to su:

1. ukupno vreme za određen deo programa
2. *calling context tree* (CCT) format podataka

Prvi slučaj je jednostavan i svodi se na dodavanje instrumentacije na mesto koje se posmatra. To je obično jedna (npr. u slučaju merenja zauzeća memorije) ili dve instrukcije (npr. u slučaju merenja trajanja izvršavanja metode)

U drugom slučaju (detaljno opisan u [Ammons97]), prikazuje se kontekst poziva – kompletno stablo poziva metoda sa naznakama izmerenih vrednosti za svaku granu u stablu. Instrumentacija je ovde kompleksna jer, pored beleženja izmerenih vrednosti (koje se izvodi za svaku metodu pojedinačno, na isti način kao u prvom slučaju), moraju da se beleže i potrebni podaci da se dobije korektno stablo poziva metoda. Ovaj format podataka pruža najdetaljniju sliku jer se razdvajaju merenja za pozive jedne procedure sa različitih mesta. JFluid pomoću dodatnih promenljivih rešava problem instrumentacije klasa koje se pozivaju i u stablu poziva i u okviru koda koji vrši instrumentaciju, da bi se izbegla beskonačna rekurzija. Problem sa ovim pristup je količina informacija koja nastaje. u procesu profajliranja. Takođe, izuzeci se ovde posebno obrađuju da ne bi promakao neki izlazak iz metode i da se dobiju nekorektni rezultati.

Kreiranje CCT-a pomoću *sampling* profajlera je prikazano u [Zhuang06]. Alat obezbeđuje kreiranje CCT-a pomoću JVMPI uz *overhead* od 80-90%. Mehanizam je dodatno poboljšan upotrebom adaptivnog uzorkovanja (eng. *adaptive sampling*).

Alat radi na sledeći način: u trenutku kada se vrši profajliranje, profajler prolazi kroz ceo stek (eng. *stack*) i beleži gde se nalazi. Nakon toga, vrši se beleženje niza poziva (ovo autori nazivaju *burst*) tokom kratkog perioda vremena (koji autori nazivaju *burst length*). Ako bi se izvršavao samo *burst*, dobio bi se neprecizan CCT. Adaptivno uzorkovanje isključuje beleženje niza poziva za koji se procenjuje da će da budu redundantni (na osnovu istorije) i tako smanjuje *overhead*.

Iako je veoma zastupljena, upotreba JVM TI za instrumentaciju Java *byte*-koda se pokazala nepogodnom iz više razloga. U nekoliko radova ([Reiss08][Zhuang06]) dokumentovano je da JVM TI izaziva nestabilnost, da dolazi do blokade i naglog prekida rada virtuelne mašine. Drugi problem je upotreba C/C++ programskog jezika za kreiranje agenata, što stvara probleme Java programerima. Treći problem se direktno nastavlja na drugi: komunikacija sa agentima napisanim u C/C++ programskom jeziku mora da se izvršava preko *native* poziva, što unosi dodatno usporenje u sistem. Zbog svega ovoga traži se način da se instrumentacija ne vrši pomoću JVM TI API-ja nego na neki drugi način.

Komorium profajler, predstavljen u [Binder06], vrši transformaciju programa tako da se u toku izvršavanja vrši brojanje izvršenog *byte*-koda. Ovde je reč o profajleru koji vrši profajliranje uzorkovanjem. Kada brojač koda dostigne određenu vrednost, pokreće se agent koja pristupa steku i uzima uzorak. Odmah se vrši i kreiranje kontekstnog stabla i beleži količina izvršenog *byte*-koda. Transformacija programa se vrši pomoću alata koji ubacuje potrebne instrukcije u programski kod. Pomoću ovih instrukcija kreira se programski stek koji se periodično uzorkuje, jer standardni stek nije dostupan iz Java programa.

Prednost ovog pristupa je što se agenti pišu u čistom programskom jeziku Java, tj. ne zahteva nikakve izmene na JVM. Pošto se broji količina izvršenog *byte*-koda (ne meri se vreme), dobijeni rezultati su potpuno nezavisni od JVM, operativnog sistema i računara na kom se program izvršava. Takođe, dobijeno stablo je precizno, jer Komorium beleži potpuni potpis metoda, što nije podržano standardnim Java API-jima koji ne razlikuju metode koje se preklapaju. Komorium je fleksibilan jer dozvoljava izmenu parametara profajliranja – može da se zada deo koda koji se prati i može da se zada vrednost koja aktivira agenta, koji uzima uzorak. Kao nedostatke Komoriuma, autori navode nemogućnost profajliranja *native* koda (jer ne može da se izbroji kao *byte*-kod), kao i Java koda koji se poziva u okviru *native* koda. Alat nije dostupan za proveru rezultata.

Postoji nekoliko alata koji vrše *byte*-kod instrumentaciju. Najpoznatiji je BCEL (eng. *Byte Code Engineering Library*) [BCEL], koji je razvijen uz sponzorstvo *Apache Software Foundation* [ASF], a u okviru *Jakarta* projekta [Jakarta]. Biblioteka omogućava, na binarnom nivou, analizu, kreiranje i manipulaciju datotekama sa Java klasama. Klase se predstavljaju objektima koji sadrže sve informacije o klasi (npr. metode, atribute, *byte*-kod instrukcije). Klase mogu da se menjaju u toku izvršavanja programa, ali postoji i mogućnost da se kreiraju potpuno nove klase.

BCEL se koristi za de-kompajliranje, refaktorizaciju (izmena koda programa bez izmene njegove funkcionalnosti) i obfuskaciju (izmena programskog koda tako da on postane nerazumljiv običnom čoveku - u nekim slučajevima i kompajlerima, ali da zadrži funkcionalnost). Upotrebljava se i za implementaciju novih koncepata, kao što je aspekt-orijentisano programiranje (eng. *Aspect Oriented Programming* – AOP) [Kiczales96]. Najpoznatiji AOP alat koji koristi BCEL je AspectJ [AspectJ]. Pošto ima mogućnost ubacivanja programskog koda, BCEL je pogodan za instrumentaciju, pa samim tim i za profajliranje.

ASM biblioteka [ASM] je projekat konzorcijuma *ObjectWeb* (sada u okviru OW2 [OW2]). Ima slične mogućnosti kao BCEL. Pored instrumentacije, koristi se za implementaciju novih koncepata u programskom jeziku Java, kao što je dinamički jezik *Groovy* [Skeet07], a koristio se u alatu *AspectWerkz* [AspectW] za AOP.

Javassist [Javassist] biblioteka je projekat JBoss-a, koji se koristi za AOP, refleksiju i implementaciju udaljenih poziva (eng. *remote method invocation*) bez generisanja *stub* klasa.

Iako imaju velike mogućnosti, biblioteke za manipulaciju *byte*-kodom su dosta komplikovane za upotrebu. Upotreba ovih biblioteka zahteva poznavanje specifičnih detalja JVM (praktično se svodi na asemblersko programiranje za JVM). Ovakav pristup može da izazove probleme i nestabilnost u radu i pored toga što ovi alati imaju zaštitu i proveru da li je dobijeni *byte*-kod napisan prema pravilima ili ne [BCEL, Javassist, Hoorn12].

S obzirom da se ove biblioteke već koriste u okviru AOP alata, mnogo je jednostavnije za razvoj koristiti AOP umesto direktne manipulacije *byte*-kodom. Razlog je jednostavan – aspekti su čitljiviji i lakše se održavaju i menjaju, nego programi koji koriste neku od navedenih biblioteka.

Upotreba AOP nije ograničena samo na programski jezik Java. U [CSharpAOP] je prikazana upotreba AOP u C#, a u [Lafferty03] alat Weave.NET pomoću koga se pomoću XML-a definišu aspekti za sve programske jezike .NET razvojnog okruženja. Weave.NET koristi sintaksu XML-a koja je slična sintaksi AspectJ alata. Pored Weave.NET, postoje i drugi (*open source*) alati koji omogućuju upotrebu AOP u .NET okruženju, npr. [Loom], [AspectDNG] i [Dotspect].

AOP alati postoje i za Cobol ([COBOLAOP]), Ruby ([Aquarium]) i PHP ([APDT]).

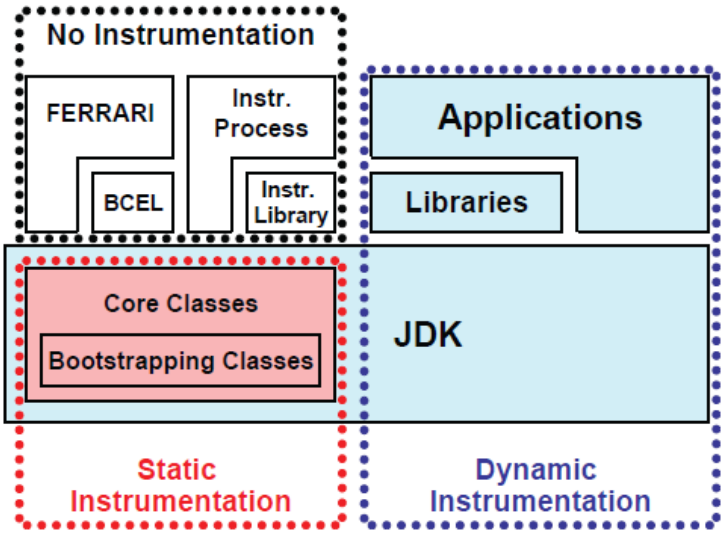
Mogućnost upotrebe aspekt-orijentisanog programiranja, konkretno alata AspectJ, u procesu profajliranja prikazana je u [Pearce07]. Dok se program izvršava, on ulazi u određene tačke, tj. izaziva određene događaje.

Te tačke se u AOP nazivaju *join point*. Pomoću AOP-a, moguće je dodati programski kod pre, posle i oko tih tačaka. Taj programski kod se u AOP naziva *advice*. Tačke mogu da predstavljaju razne konstrukte u programskom jeziku: poziv metode, poziv konstruktora, izvršavanje metode, pristup atributu, itd. Povezivanje *join point*-a sa *advice*-om se vrši preko *pointcut*-a. Pored mogućnosti ubacivanja programskog koda, AspectJ nudi još i mogućnost otkrivanja koja tačka je pokrenula neki *advice* (nešto slično *this* ključnoj reči u programskom jeziku Java), kao i dodavanja novih metoda i atributa postojećim klasama. U radu je razmotreno profajliranje dinamičkog zauzimanja memorije (eng. *heap profiling*), životnog veka objekata i vremena izvršavanja.

Sprovedeno testiranje pokazuje da je profajliranje pomoću AspectJ alata dovoljno fleksibilno da se pomoću njega vrši profajliranje različitih podataka, uz ograničenja opisana u radu – nedostatak rada sa sinhronizovanim blokovima, nemogućnost prepoznavanja alokacije nizova – i još neke probleme koji su u međuvremenu rešeni u novijim AspectJ implementacijama (npr. upredanje aspekata u toku učitavanja – eng. *load time weaving*). Studija prikazana u [Marwede09] pokazuje da profajleri koji su zasnovani na AOP pružaju dobru osnovu za razvoj alata za profajliranje, jer profajleri zasnovani na JVM TI, ipak unose veće kašnjenje i grešku merenja.

Jedan od osnovnih nedostataka instrumentacije pomoću AOP je taj što lako može da dođe do beskonačne petlje ako se u okviru aspekta koristi klasa koja je instrumetirana. AspectJ ima zaštitu od ove pojave pri kompajliranju, ali ako se upredanje vrši pri učitavanju klasa (eng. *load time weaving*) dolazi do greške. Drugi problem je nedostatak mogućnosti dinamičkog upredanja – sav kod mora da bude spreman pre izvršavanja i nema mogućnosti za izmene nakon što izvršavanje počne. Ovo je posebno značajno kod profajliranja aplikacija koje se dugo izvršavaju, jer aplikacija mora da se prekine da bi se izmenili parametri profajliranja.

Okruženje FERRARI [Binder07] rešava ove probleme i može da izvrši instrumentaciju proizvoljnog Java koda. Pošto dinamička instrumentacija može da unese kašnjenje u sistem, FERRARI vrši statičku instrumentaciju osnovnih Java klasa koje se učitavaju pre instrumentacije, dok se ostale instrumentiraju dinamički (Slika 2.3).



Slika 2.3 Prikaz odnosa dinamičke i statičke instrumentacije pri upotrebi FERRARI okruženja (preuzeto iz [Binder07])

FERRARI je pogodan za kreiranje preciznih profajlera. U radu je prikazano kreiranje stabla poziva i mogućnost profajliranja pomoću JP profajlera. Rezultati, dobijeni pomoću SPEC JVM 98, pokazuju da u odnosu na neprofajliranu aplikaciju, JP unosi kašnjenje od oko 50%. Dobijeno kašnjenje je manje u odnosu na profajlere koji koriste JVM TI ili JVM PI.

Uz oslonac na FERRARI okruženje, razvijen je HotWave [Villazon09a]. HotWave proširuje AspectJ i omogućuje da dobijeni kod bude u skladu sa mehanizmom za zamenu koda (eng. *hotswapping*). Treba napomenuti da HotWave direktno ne podržava *around advice* koji postoji u AspectJ, nego ga implementira pomoću kombinacije *before* i *after advice*-a. U radu je prikazana primena HotWave alata u adaptivnom profajliranju. Korisnik može da posmatra izvršavanje aplikacije iz IDE-a i da menja parametre procesa profajliranja, bez potrebe da zaustavlja rad aplikacije. Na ovaj način može da se smanji kašnjenje sistema i greška merenja jer se vrši profajliranje samo onog dela softvera koji korisnik želi da prati. Izmena parametara, praktično, zahteva izmenu aspekata, pa se HotWave koristi za uklanjanje već upredenih i upredanje novih aspekata (eng. *reweaving*). Alat prepoznaje koje klase je potrebno ponovo upredati i ne dira one koje ne treba. Detaljniji prikaz ovog postupka dat je u [Villazon09b].

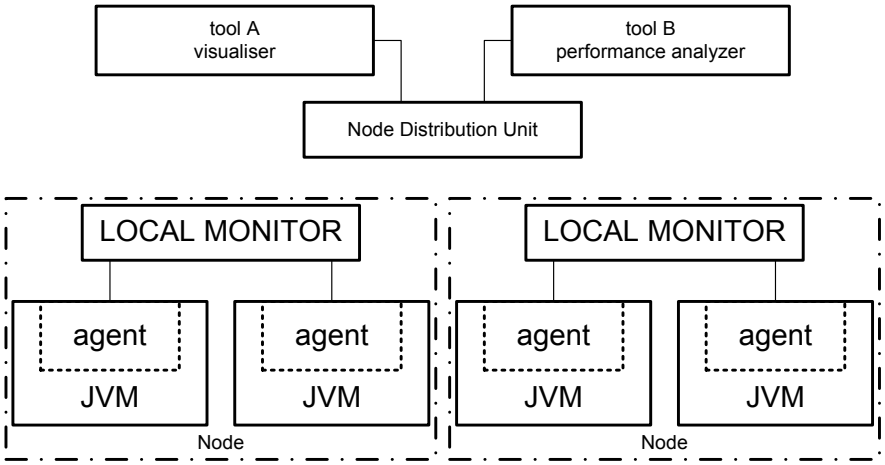
Sistem nije dostupan, pošto je još uvek u razvoju.

Alati i okruženja prikazani u [Pixie89], [Larus94] i [Srivastava94] namenjeni su za više-procesorske računare, tj. aplikacije koje se izvršavaju na ovim računarima, ali ne i distribuirane aplikacije koji se izvršavaju na više računara istovremeno. Kao što je rečeno u uvodu, ove aplikacije donose dodatne probleme, pa se oni moraju dodatno razmotriti.

U [Bubak03] dati su zahtevi koje alat za praćenje distribuiranih aplikacija:

- alat treba da podržava aplikacije koje se izvršavaju na jednoj ili više JVM
- alat treba da uzima u obzir da je Java objektno-orijentisan jezik
- alat treba da omogući korisnicima da rezultate mogu da analiziraju i u toku rada aplikacije (eng. *on-line*)
- analiza treba da je omogućena na više nivoa – od globalnog pogleda na aplikaciju do pogleda na najsitnije detalje
- alat treba da je proširiv i da je ta proširenja moguće razumno lako implementirati

Predložena arhitektura okruženja za nadgledanje distribuirane Java aplikacije je data na slici 2.4.



Slika 2.4 Predložena arhitektura okruženja za praćenje distribuirane Java aplikacije

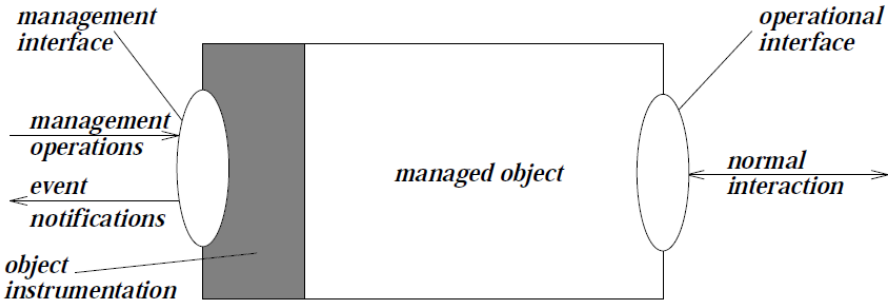
Na vrhu su alati koje krajnji korisnik koristi za analizu i nadgledanje. *Node distribution unit* (NDU) je računar na kom se izvršavaju korisnički alati i njegov zadatak je da svaki zahtev od strane alata razbije na komande koje će da prosledi odgovarajućem čvoru. Distribuirani deo ove arhitekture predstavljen je sa *node local monitor*-ima (NLM). NLM nadzire lokalni deo

sistema i komunicira sa JVM *local monitor*-ima (JVMLM) koji kontroliše jednu JVM. Komunikacija sa JVM može da se odvija preko JVM PI API-ja.

Ceo sistem predstavlja Java-orijentisano proširenje OMIS specifikacije [Ludwig97]. OMIS specifikacija definiše standardizovan interfejs između korisničkih alata i sistema koji vrši nadgledanje. OMIS nije ograničen na jedan skup alata – može da se koristi za alate za nadgledanje, profajliranje, upravljanje resursima.

I u [Schade96] se koristi pristup po kom se pravi univerzalni interfejs za povezivanje različitih alata sa različitim aplikacijama. Distribuirani sistemi zahtevaju posebne upravljačke podsisteme, koji na osnovu podataka dobijenih (najčešće) instrumentacijom, šalju određene upravljačke komande. Zadaci koji se postavljaju pred ovakve sisteme su: opštost (u odnosu na tip aplikacije), proširivost i fleksibilnost (mogućnost migracije komponenti). Ovakva infrastruktura distribuirane aplikacije i njenog upravljačkog sistema pogodna je za vršenje profajliranja aplikacije, gde se procesom profajliranja upravlja isto kao što se upravlja samom aplikacijom.

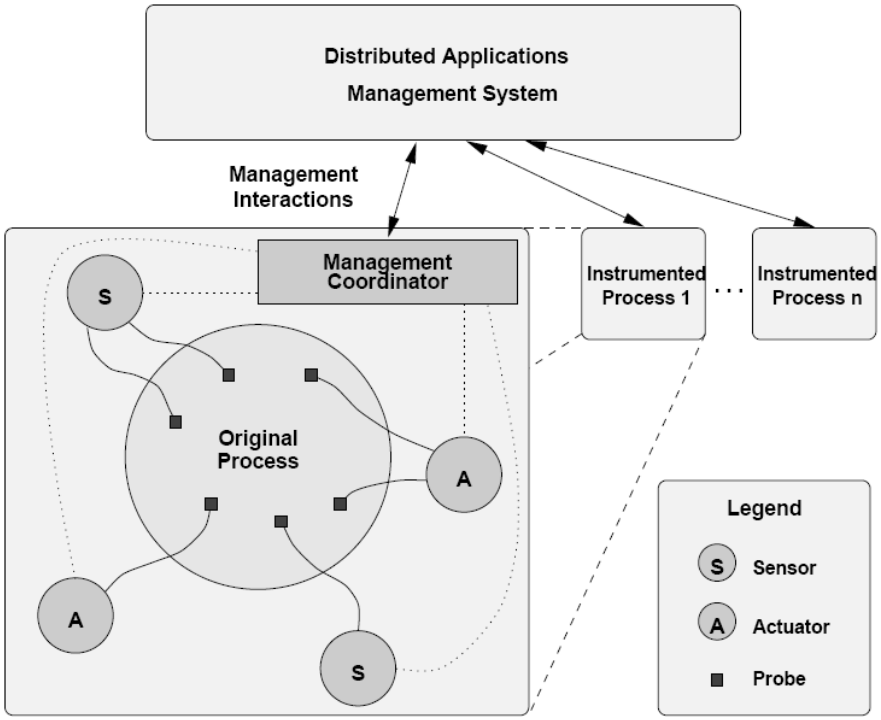
Struktura jednog objekta upravljanja data je na slici 2.5.



Slika 2.5 Struktura objekta upravljanja (preuzeto iz [Schade96])

Objektu se pristupa preko upravljačkog interfejsa, a on preko operacionog interfejsa (primarnog interfejsa) komunicira sa ostatkom sistema. Informacije o svom radu, upravljani objekat šalje mehanizmom notifikacija. Ova arhitektura definisana je 1996, tj. pre prve revizije JMX specifikacije koja se pojavila 1999 [JMX], ali čiji elementi se mogu prepoznati u ovoj arhitekturi.

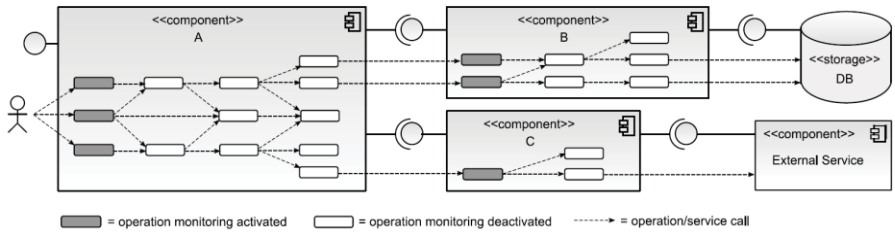
Sličan koncept prikazan je u [Katchabaw97]. Upravljačko okruženje je prikazano na slici 2.6.



Slika 2.6 Upravljačko okruženje (preuzeto iz [Katchabaw97])

U radu je prikazana arhitektura i razmotreni pristupi u instrumentaciji. Prikazani je koncepti sonde – komponenta koja je ugrađena u proces. Preko sondi senzori vrše nadgledanje izvršavanja procesa, a aktuatori se koriste da izvrše upravljanje. Koordinator (eng. *Management Coordinator*) kontroliše rad senzora i aktuatora i prima komande od upravljačkog sistema.

U radu [Ehlers11], autori predstavljaju još jedan sistem za detekciju anomalija, koji je takođe zasnovan na Kieker okruženju i posmatranju stabla poziva, kao i sistem opisan u ovoj tezi. Pri praćenju, posmatraju se pojedinačne komponente sistema. Pre početka postavljaju se softverske sonde u metode komponenata koje želimo da pratimo. Na početku se prate samo metode koje čine interfejs komponenata, tj. metode koje su na najvišem nivou stabla poziva (slika 2.7).



Slika 2.7 Početno stanje softverskih sondi (preuzeto iz [Ehlers11])

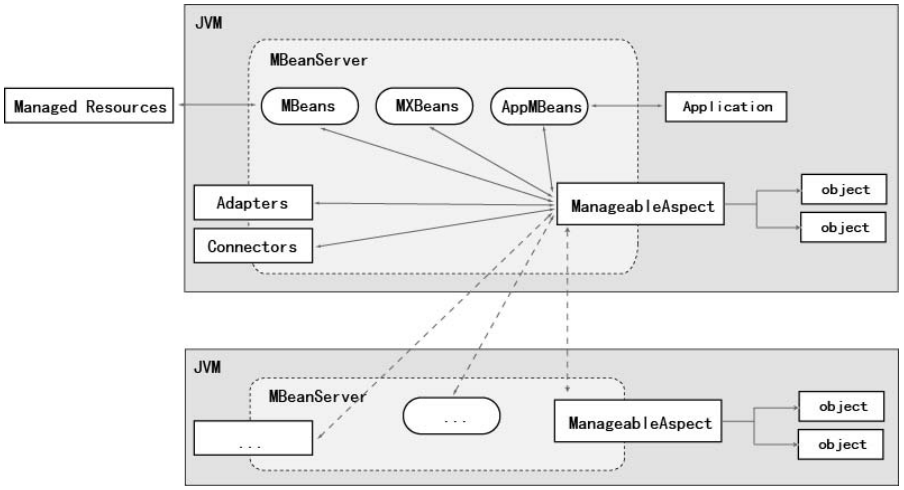
Ako se, tokom praćenja, primeti odstupanje od očekivanih vrednosti, zadatih pomoću OCL-a, uključuje se sonde za praćenje na sledećem nivou stabla poziva u okviru komponente, itd. Uključivanje i isključivanje može da se vrši i ručno, putem proširenja Eclipse okruženja.

Većina modernih distribuiranih aplikacija zasnovana je na JavaEE [JavaEE] (ranije J2EE) tehnologiji. U osnovi većine Java aplikacionih servera, na kojima se izvršvaju JEE aplikacije, nalazi se upravo JMX. Pomoću JMX tehnologije se na standardizovan način pristupa različitim elementima jednog sistema. Potrebno je samo komponente tog sistema opisati *MBean*-ovima. *MBean* može da se upotrebi za podešavanje konfiguracije aplikacije, za dobijanje različitih vrsta podataka o stanju aplikacije i slanje obaveštenje (notifikacija) o stanju aplikacije.

Jedna od mogućnosti za kreiranje alata za profajliranje distribuiranih aplikacija data je u [Liu04]. Autori su pokušali da naprave kombinaciju aspekata i *MBean*-ova – upravljane aspekte (eng. *Manageable Aspect*). Ovi aspekti predstavljaju aspekte kojima može da se upravlja kao standardnim MBean-ovima. Arhitektura sistema koji koristi upravljane aspekte data je na slici 2.8.

Kao MBean, *ManagableAspect* komponente koriste *MBean*-ove koji se nalaze u okviru *MBeanServer*-a i prikupljaju podatke od njih. Podaci mogu da se prikupljaju od *MBean*-ova koje kreira aplikacija (*AppMBeans*), od standardnih platformskih *MBean*-ova (*MXBeans*, pog. 3.7.2), kao i od bilo kojih drugih *MBean*-ova. S druge strane, kao aspekt, *ManagableAspect* komponenta presreće sve pozive koji su definisani u aspektu, kao *joinpoint*-i.

Pristup i upravljanje *ManagableAspect* komponentama može da se vrši preko bilo koje JMX konzole, putem adaptera i konektora koji postoje u svakom *MBeanServer*-u.



Slika 2.8 Arhitektura sistema koji koristi aspekte kao MBean-ove (preuzeto iz [Liu04])

U radu su predložena i proširenja AspectJ sintakse tako da se klase koje predstavljaju *MBean* anotiraju kao aspekti. Razmotrena je i mogućnost profajliranja memorije i programskih niti preko standardnih platformskih MBean-ova (tj. MXBeans komponente na).

Arhitektura prezentovana u ovom radu nije implementirana. Ipak, ona daje dobru osnovu za dalja istraživanja jer kombinuje JMX (koji se nalazi u osnovi svih distribuiranih JEE aplikacija) sa AOP (koji je najpogodniji za instrumentaciju i profajliranje Java aplikacija).

U [Ehlers11] prezentovan je adaptivni sistem koji je, kao i sistem prikazan u ovoj tezi, baziran na analizi stabla poziva i Kieker okruženju. Za svaki čvor u stablu poziva poredi se dobijeno vreme odziva sa očekivanim vremenima. Očekivano vreme se računa na osnovu prethodnih vrednosti. Automatska promena parametara praćenja, tj. uključivanje dodatnih čvorova u praćenje, vrši se na osnovu očekivanih vrednosti, trenutnih parametara praćenja i zadatih pravila. Za definisanje pravila koristi se OCL [OCL]. U [Marwede09] prikazana je automatska lokalizacija problema zasnovana na pristupu koji traži korelaciju anomalija sa zavisnostima između komponenti. I ovde je upotrebljen Kieker za praćenje, ali praćenje nije adaptivno.

RaceTrack alat za pronalaženje uzroka nekonzistentnih rezultata (eng. race condition) u .NET aplikacijama prikazan je u [Yu05]. Alat vrši praćenje rada softvera i traži sumnjive obrasce. Prati se objekti, a po potrebi uključuje se i praćenje pristup njihovim atributima. Implementiran je

modifikovanje .NET virtualne mašine - common language runtime (CLR). Ovo može da izazove probleme pri prelasku na noviju verziju CLR-a i zahteva da se modifikovana verzija CLR-a distribuira zajedno sa aplikacijom koja se prati, umesto da se alat distribuira, kao što je slučaj sa sistemom opisanim u ovoj tezi.

Pinpoint sistem [Chen02] locira komponente koje bi mogle da izazovu problem. Traži se korelacija između problema na nižim nivoima i grešaka u radu koji se javljaju na visokim nivoima.

Koristi se modifikovana Java EE platforma koja skuplja i analizira klijentske pozive. Sistem ne beleži performanse i više vrši se koristi za lokalizaciju problema.

U nekim radovima koristi se pristup "crna kutija" za pronalaženje komponente koja izaziva probleme, ali ne i tačnog razloga u okviru komponente. U [Aguilera03] prikazan je pristup koji prati poruke koje se razmenjuju između komponenti i traži veze između poruka i problema u performansama. Alat PeerPressure prikazan u [Wang04] poredi rezultate "zdravih" i "sumnjivih" računara pomoću statističkih metoda, da bi pronašao probleme.

U vrlo malo radova je razmotreno pitanje uticaja sistema za praćenje na rad celog sistema. Za Kieker je to urađeno u [Hoorn12]. U [Dimitriev03] su testirane performanse sistema JFluid pomoću SPEC jvm98 alata [SPECjvm98]. Pogoršanje vremena odziva pri upotrebi ovog sistema ide od 1% za zadatke koji traju dugo (npr. pristup bazi podataka) do 5000% za brze zadatke (npr. kompresija datoteka). Korisnici ovo mogu da smanje tako što će da biraju da se prate samo neki delovi softvera. U [Govindraj06] prikazana je mogućnost praćenja softvera pomoću AOP. Očekivano produženje vremena odziva ide od 1-10% što se uklapa u analizu Kieker sistema koji, takođe, koristi AOP.

Od komercijalnih sistema jedino DynaTrace daje podatke od uticaju. Oni navode da je maksimalan uticaj manji od 5%.

Autori retko navode hardver i softver na kojima je izvršeno merenje, a koji značajno utiču na rezultate.

2.2 Definisanje nivoa kvaliteta softverskog servisa

Ugovor o nivou kvaliteta servisa [SLAnet] (eng. *service level agreement* – u daljem tekstu SLA) predstavlja deo ugovora između korisnika servisa (eng. *service consumer, client*) i onoga ko pruža uslugu (eng. *service provider*). SLA predstavlja specifikaciju po kojoj se vrši stalna identifikacija, nadzor i provera nivoa IT servisa [Meyer05]. Može da bude i u pravno obavezujućem i u neformalnom obliku. Po ovom ugovoru ne

garantuje se samo pružanje usluge nego i njen kvalitet u skladu sa parametrima. Ugovor [SLAZone] definiše meru parametara servisa (kao što su npr. vreme odziva, protok, memorijsko zauzeće, opterećenje procesora), zahtevane nivoe kvaliteta servisa (eng. *quality of service* – u daljem tekstu QoS) u zadatoj metrici, kao i upravljanje servisom i reakcije na probleme u zadovoljavanju zadatog nivoa kvaliteta (akcije za oporavak servisa, kazne za pružaoca usluge). SLA treba da sadrži i vremenske odrednice kao što su period za koji se vrši praćenje i učestalost merenja.

Životni ciklus jednog SLA [Sturm00] je prikazan na slici slici 2.9.

Slika 2.9 Životni ciklus SLA (prema [Sturm00])

Životni ciklus SLA počinje definisanjem ugovora. Definisani ugovor se prosleđuje pružaocu usluge na izvršavanje. U ovoj fazi, praktično vrši se raspodela zaduženja i počinje primena ugovora. Treća faza predstavlja praćenje rada servisa u skladu sa ugovorom. Prikupljeni podaci se analiziraju i koriste se za: 1) poboljšanje samog servisa (otklanjanje problema u radu, optimizacija rada – faza 5) i 2) preradu ugovora (faza 6).

U [Trienekens04] su definisani principi specifikacije SLA: princip kontinuiteta, *pit*/*shell* princip i princip kvaliteta procesa servisa i servisnih objekata. Princip kontinuiteta definiše životni ciklus SLA, njegovo definisanje, praćenje i preradu sadržaja SLA. *Pit*/*shell* princip definiše: *pit* – objekat servisa/servisni proces koji je od interesa i *shell* – okruženje u kom se koristi objekat/servis. Treći princip razgraničava kvalitet procesa servisa – koji je odgovornost pružaoca usluge – i kvalitet servisnih objekata – koji su odgovornost korisnika usluge i nad kojima se servis vrši.

U [Paschke06] izvršena je kategorizacija SLA i mera koje se koriste u SLA. Identifikovani su standardni elementi svakog SLA: tehnički, organizacioni i pravni. U tehničke elemente spadaju opis servisa, objekti

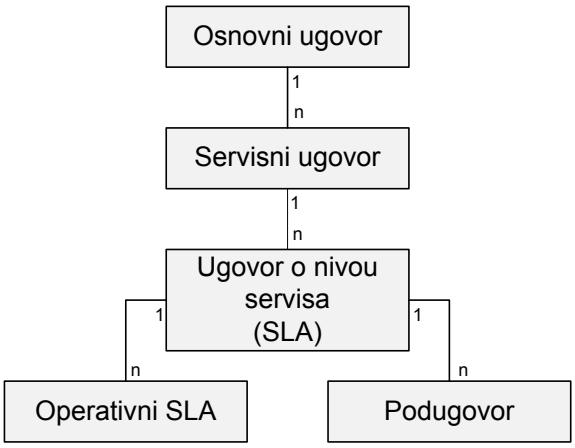
koji postoje u servisu, QoS parametri, mere koje se koriste i akcije koje se sprovode. Organizacioni elementi su odgovornosti, praćenje i izveštavanje, održavanje/servisiranje i upravljanje izmenama. U pravne elemente spadaju obaveze u saradnji, pravne obaveze, dodatna prava i načini plaćanja.

Izvršena je kategorizacija mera prema standardnim IT objektima: hardver, softver, mreža, masovna memorija i podrška. Deo uobičajenih QoS parametara i odgovarajuće mere su dati u tabeli 2.1.

Tabela 2.1 Primeri mera za različite QoS parametre (prema [Paschke06])

Objekat	**QoS parametar**	**Mera (jedinica)**
Softver	brzina odgovora	vreme (sekunde)
Softver	dostupnost	vreme (sekunde)
Hardver	brzina izvršavanja instrukcija	broj instrukcija u vremenu (broj/sekunda)
Hardver	radne stanice	broj računara (broj)
Mreža	kašnjenje mreže	vreme (milisekunde)

Autori su izvršili i kategorizaciju SLA dokumenata. Prema nameni, dokumenti se dele na osnovne (osnova za ostale dokumente), servisne (može da obuhvata više drugih SLA), obične ugovore o nivou servisa, operativne SLA (ugovor sa partnerima u okviru organizacije sa kojima se zajedno radi na nadređenom SLA) i pod-ugovore (ugovor sa partnerima van organizacije sa kojima se zajedno radi na nadređenom SLA). Odnos između tipova ugovora prema ovoj podeli je dat na slici 2.10.



Slika 2.10 Odnos između različitih tipova ugovora prema nameni (prema [Paschke06])

Prema okviru primene postoje interni ugovori (više neformalni dogovori, nego ugovori), ugovori u okviru organizacije (ugovori između organizacionih jedinica u okviru jedne organizacije), eksterni ugovori (ugovori između korisnika i pružaoca usluga) i višeslojni ugovori (ugovori koji definiše odnose i prema trećoj strani).

Treća podela je prema raznovrsnosti primene. Ugovori mogu da budu standardni, prošireni (standardni ugovor sa dodatnim dodatnim podugovorima), individualni (posebno prilagođeni ugovori) i fleksibilni ugovori (mešavina standardnih i pojedinačnih ugovora).

Istraživanje prikazano u [Tebbani06] pokazuje nedostatak postojanja formalnog jezika za definisanje SLA. Većina SLA se piše običnim (govornim) jezikom, što nije pogodno za automatizaciju procesa upravljanja nivoom servisa. Autori dalje daju specifikaciju GXLA – jezika za definisanje SLA. GXLA je XML shema zasnovana na GSLA – opštem SLA modelu. Osnovni elementi GSLA su:

1. *role* – uloge u procesu implementacije SLA – ko je za šta zadužen)
2. *party* – učesnik (ujedno i potpisnik) u SLA
3. *service package* (SP) – objekti od kojih se sastoji SLA
4. *service package objective* (SPO) – cilj koji neko od učesnika treba da ispuni u vezi sa zadatim objektom

Pravila koja definišu SPO se definišu pomoću predikata (za zahtevani QoS parametar se definišu određene vrednosti) i garancije (koji učesnik mora da zadovolji SPO i koje akcije se primenjuju).

Pomoću jezika za opis SLA za veb servise (eng. *web service level agreement* – WSLA) [Keller09] definiše se koji nivo usluge se očekuje i koje mere treba da se preduzmu u slučaju da dođe do odstupanja. Definišu se mere i način merenja, kao i zaduženja u samom procesu praćenja. WSLA je zasnovan na XML-u i proširiv je, tj. može da se prilagodi različitim platformama, tehnologijama, mogu da se definišu nove operacije i mere. Jedan WSLA dokument sadrži:

1. opis ugovornih strana (eng. *parties*) i njihovih uloga; svako od njih ima svoj interfejs sa akcijama koje može da izvršava i preko kog komunicira sa ostalim učesnicima
2. specifikaciju SLA parametara pomoću odgovarajućih mera; ovde se definiše kako se vrši merenje i ko vrši merenje
3. definiciju obaveza učesnika – formalan opis garantovanih uslova koji treba da se prate u zadatom periodu

SLAng [Lamanna03] standard je još jedan standard koji za cilj ima automatizaciju procesa upravljanja nivoom servisa. Zasnovan je na Meta Object Facility [MOF] standardu. Za definisanje SLAng dokumenata mogu

da se koriste različiti jezici, npr. *human-usable textual notation* - HUTN [HUTN] ili *object constraint language* - OCL [OCL].

WS-Agreement [Oldham06] jezik je odobren od strane Open Grid Forum-a. Specifikacija jezika se sastoji od XML sheme za definisanje ugovora, sheme za definisanje šablona ugovora i skupa operacija koje mogu da se koriste tokom celog životnog ciklusa SLA (od kreiranja SLA, praćenja u svakom koraku ciklusa do isticanja upotrebe ugovora).

Kreiranje dokumenata po ovim specifikacijama, zahteva dosta rada i kao rezultat daje često glomazne dokumente. Samo WSLA iza sebe ima ozbiljnu podršku u vidu okruženja za kreiranje dokumenata.

2.3 Predviđanje performansi aplikacije

Performanse softvera su od fundamentalnog značaja u distribuiranim arhitekturama koje su danas u sve široj upotrebi (npr. *cloud*, *grid*, ...). Zbog toga je razvijena oblast upravljanja performansama (eng. performance engineering). Na upravljanje performansama danas se gleda kao na deo životnog ciklusa softvera i uključeno je u ITIL biblioteku. Ciljevi upravljanja performansama su:

- uvećanje prihoda kroz osiguranje da sistem obrađuje zahteve u predviđenom vremenu,
- isključivanje odbacivanja sistema usled problema sa performansama
- isključivanje potrebe za preradom sistema usled problema sa performansama
- izbegavanje nabavke novog hardvera u cilju ostvarivanja zadatih performansi
- redukcija cene održavanja zbog problema sa performansama

Prema ITIL-u, upravljanje performansama se koristi u sledećim procesima:

1. upravljanje nivoom servisa – cilj je da performanse softverskog servisa budu na takvom nivou da servis ispunjava zadate nivoe
2. upravljanje kapacitetom – upravljanje performansama ovde za cilj ima da se osigura da softverski servis ispunjava svoje zadatke u zadatim okvirima performansi. Na osnovu podataka dobijenih u praćenju, predviđa se pod kojim opterećenjem on više ne može da ispunjava zahteve u okviru zadatih performansi i planiraju se akcije koje će obezbediti da obaveze budu ispunjene.
3. upravljanje problemima – upravljanje performansama se ovde koristi u traženju izvora problema performansi. Vrši se podešavanje sistema, rekonfigurisanje parametara i prerada aplikacije u cilju poboljšanja performansi.

Što se tiče podataka koji se koriste u upravljanju performansama softverskog servisa oni moraju da daju realnu sliku servisa. To znači da prikupljanje podataka mora da bude obavljeno tako da što manje utiče na performanse sistema i da bude obavljeno pod stvarnim opterećenjem. Merenje performansi u toku razvoja aplikacije nije dovoljno. Dobijeni podaci mogu da se koriste za raspoređivanje resursa i procesa [Jarvis06a], a sve u cilju ispunjavanja funkcija u okviru zadatih performansi.

Aktuelizacija pitanja performansi kompleksnih softverskih servisa dovela je do razvoja pristupa koji se koriste da bi se obezbedile zadate performanse. Jedan takav pristup je opisan u [Jarvis06b]. Opisana je metodologija koja obezbeđuje informacije o performansama tokom celog životnog ciklusa softverskog proizvoda. Priroda pristupa je takva da je moguće izvršiti analizu pre i posle implementacije. Analiza je optimizovana sa ciljem da se što brže izvršava i da što manje opterećuje sistem na kom se izvršava, tako da je moguće da se vrši i tokom izvršavanja aplikacije.

Na osnovu prikazane metodologije razvijen je alat PACE [Nudd00].

Nedostatak ovog pristupa je taj što se u razvoj aplikacije uključuje još jedna metodologija, što može da dodatno optereti razvojni ti i produži razvoj softvera.

Umesto uključivanja nove metodologije u razvoj, neki autori se odlučuju za tehnike mašinskog učenja.

Jedna od prvih primena neuronskih mreža za predviđanje pouzdanosti softvera je data u [Karunanithi92]. Pokazano je da je modelima zasnovanim na neuronskim mrežama dovoljno da imaju informaciju o ponašanju softvera (u odgovarajućim jedinicama mere) da bi ostvarile zadovoljavajuću preciznost modela. Mreže se modeluju tako da na osnovu datog skupa prethodnih vremena izvršavanja i broja grešaka koje su se javile, predvide broj grešaka nakon određenog vremena, tj. broja ponavaljanja testa. Pri modelovanju neuronske mreže, vodi se računa o:

- arhitekturi – mora se voditi računa o kompleksnosti mreže, jer kompleksna mreža obezbeđuje preciznost, ali zauzima više resursa i treb joj više vremena za obučavanje,
- obučavajućem skupu – vrši se izbor između opšteg obučavanja (eng. *generalization training*) i prediktivnog treninga (eng. *prediction training*), tj. bira se da li se izlazna vrednost povezuje sa odgovarajućom vrednosti ulazne ili sa prethodnom, respektivno,
- algoritmu za obučavanje – obuka se vrši obično u više iteracija i postoji više algoritama, a autori koriste *back-propagation* algoritam.

U radu [Karunanithi92] je prikazana upotreba neuronske mreže za predikciju performansi paralelizovanog softvera i za raspoređivanje

paralelnih zadataka u više-procesorskim sistemima. Pristup je implementiran za programski jezik R, ali implementacija nije dostupna.

U radovima [Kanmani04] i [Tian05] prikazana je primena neuronskih mreža u predikciji grešaka u softveru. U oba slučaja, neuronske mreže se koriste za popravljanje kvaliteta softverskog proizvoda. Na osnovu dobijenih rezultata moguće je predviđati koliko grešaka postoji u kodu, koliko linija koda je potrebno menjati da bi se greške uklonile i koliko je vremena potrebno za to.

Oba rada prikazuju pristupe koji mogu da budu korisni u razvoju softvera, jer mogu da utiču na smanjenje grešaka u kodu softvera, ali ne mogu da se koriste za predviđanje ponašanja softvera koji se izvršava i predviđanje pojavljivanja stanja greške ili prekida rada.

Pristup u kom se koriste evoluirajuće neuronske mreže prikazan je u [Ipek05]. Upotrebljen je genetski algoritam pomoću kog se optimizuje arhitektura mreže. Optimizuje se broj ulaznih neurona i broj neurona u skrivenom sloju. Pristup se primenjuje u razvoju, tj. u fazi testiranja softvera.

U radu [Thwin05] autori predlažu kako da se smanji uticaj ekstremnih vrednosti na rezultat predikcije pomoću neuronske mreže. Ekstremne vrednosti koje postoje u skupu izmerenih vrednosti nastaju usled povećane aktivnosti u ostalim delovima informacionog sistema u okviru kog se izvršava aplikacija. Autori vrše stratifikaciju uzoraka (grupisanje), tako da se za svaku grupu dobiju težine. Obučavajući skup se formira tako da se elementi skupa repliciraju u skladu sa određenom težinom. Elementi sa najvećim odstupanjem u odnosu na očekivanu vrednost se ponavljaju najmanje puta, i obrnuto. Neuronska mreža se obučava *bagging* mehanizmom, srednjom vrednosti iz svake grupe, da bi se redukovala varijansa modela. Autori su prikazali primenu pristupa na jednoj paralelnoj aplikaciji. Implementacija pristupa nije dostupna, a autori kao nedostatke ovog pristupa navode potrebu za velikim brojem uzoraka koji se dugo prikupljaju.

U novije vreme, veći broj autora se orijentiše na predikciju pada aplikacije usled potrošnje resursa. Rezultati se u tom slučaju obično koriste za, već pomenutu, prevenciju pada, odnosno preventivno restartovanje aplikacije u cilju skraćivanja vremena kad je aplikacija nedostupna. Preventivno restartovanje izaziva kraće prekide u dostupnosti jer se ne gubi vreme na prijavu pada i reagovanje. Ovi prekidi su obično kraći nego prekidi koji se dogode nepredviđeno, kod kojih se dosta vremena gudi na pronalaženje i rešavanje problema.

U ovom slučaju postoje dva pristupa: periodično restartovanje i restartovanje po potrebi.

Periodično restartovanje se vrši u tačno zadatim intervalima. Obično se vrši u trenucima kada je smanjena aktivnost korisnika aplikacije.

Restartovanje po potrebi se vrši u kombinaciji sa stalnim praćenje aplikacije. Kada sistem za praćenje utvrdi da će aplikacija uskoro potrošiti dostupne resurse, vrši se preventivno restartovanje.

Oba pristupa obezbeđuju da aplikacija ima veću dostupnost nego ako se pusti da dođe u stanje pada, ali imaju i nedostatke. Periodično restartovanje će restartovati aplikaciju bez obzira na to da li ima sasvim dovoljno resursa da radi još neko duže vreme. S druge strane, restartovanje po potrebi restartovaće aplikaciju bez najave i bez obzira na to što u tom trenutku aplikaciji pristupa veći broj korisnika.

Starenje softvera u konkretnim primerima prikazano je u radu [Grottke06], kao i u [Grottke08] od istih autora. U [Grottke06] su razmotreni primeri iz "stvarnog sveta": starenje aplikacije pokrenute na Apache serveru, kao i rad Cisco Catalyst mrežnog sviča i već pomenuti problem sa raketnim sistemom Patriot, koji su imali softverske greške. Opisane su karakteristike problema i kako se oni mogu rešiti podmlađivanjem. U [Grottke08] je detaljnije razmotreno preventivno podmlađivanje Apache servera.

U [Andrzejak06] je pokušano je kreiranje opšteg modela za predikciju i preventivno podmlađivanje, u cilju automatizacije održavanja softvera. Pristup je prikazan na Apache Axis 1.3 SOAP serveru, za koji su autori ranije [Silva06] pokazali da je podložan padovima usled starenja. Pristup je zasnovan na uočavanju indikatora starenja. Cilj je da se uoče mere koje vremenom pokazuju znake starenja i zavise od vremena koje proteklo od poslednjeg pokretanja ili restarta aplikacije. Umesto vremena češće se uzima broj klijentskih zahteva, pošto se uglavnom resursi troše kad klijenti pristupaju, a ređe usled nekih vremenskih uticaja (kao što je pomenuti primer sa Patriot sistemom). Na osnovu dobijenih rezultata za izabranu meru, vrši se aproksimacija i modelovanje starenja pomoću determinističke funkcije (koja je neprekidna funkcija ili spoj više funkcija koje su neprekidne po sukcesivnim intervalima). Na osnovu dobijene funkcije može da se izvrši optimizacija rasporeda restartovanja, tj. podmlađivanja aplikacije. Optimizacija može da se izvrši na osnovu nekog od parametara u SLA ili tako da se posmatrana mera drži u nekom okviru.

Problem sa opisanim prisupom je taj što nije uvek moguće dobiti model starenja koji će dovoljno precizno opisati situaciju, a opet biti dovoljno jednostavan za primenu u optimizaciji rasporeda restartovanja. Ipak u situacijama u kojima je jedini cilj da se maksimizuje dostupnost aplikacije, ovaj pristup je dobro rešenje, pošto će dati precizan model za optimizaciju.

U [Bao05] prikazan je hijerarhijski model za analizu ponašanja i softvera i proaktivno upravljanje padovima aplikacije. Na dnu hijerarhije je model degradacije koji pomoću lanaca Markova povezuje curenje resursa sa učestanosti pada aplikacije. Na nivou iznad je proaktivno upravljanje padovima koje je takođe modelovano pomoću lanaca Markova. Model za proaktivno upravljanje se sastoji od tri čvora, slično kao na slici 1.1. Verovatnoće za prelazak iz stanja u kom sistem normalno radi u stanje pada i stanje podmlađivanja dobijaju se u modelu degradacije. Različiti oblici modela upravljanja se kreira u zavisnosti od zadatih optimizacionih ciljeva upravljanja greškama.

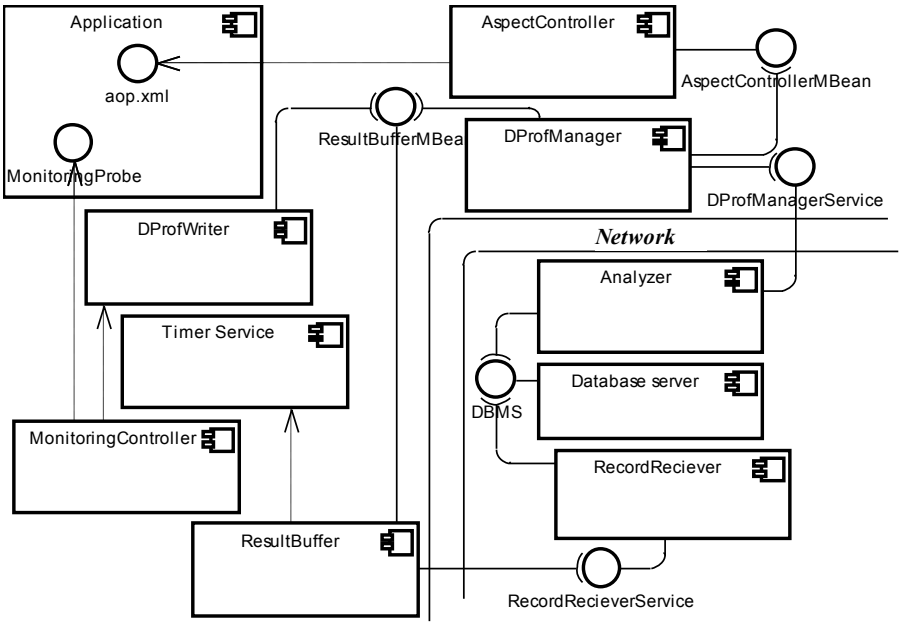
Ovaj pregled pokazuje da je teško napraviti jedan univerzalan model za predikciju ponašanja softvera. Svi prikazani modeli praktično se sastoje od više podmodela u zavisnosti od toga šta je šta se meri i šta je cilj predikcije. Jedna od mogućnosti je da se umesto kreiranja univerzalnog kreiraju konkretni modeli za konkretne situacije. U ovoj tezi su prikazana dva takva modela. Jedan model prikazuje promenu u odzivu aplikacije u zavisnosti od promene broja korisnika i serverske konfiguracije, a drugi planiranje preventivnog podmlađivanja aplikacije.

3 Specifikacija sistema za adaptivno profajliranje – DProf

DProf predstavlja sistem za adaptivno profajliranje Java aplikacija – prvenstveno distribuiranih Java EE aplikacija. Sistem je zasnovan na Kieker okruženju opisanom u [Rohr08], koje je prošireno sa dodatnim komponentama.

3.1 Arhitektura DProf sistema

Arhitektura sistema prikazana je dijagramom komponenti na slici 3.1.



Slika 3.1 Komponente DProf sistema

Instrumentacija se vrši pomoću AOP. Merenje može da se vrši pomoću standardnih mehanizama dostupnih u Java platformi ili dodatnih paketa. Prikupljanje dobijenih podataka vrši se pomoću Kieker okruženja koje je prošireno dodatnim komponentama. Podaci se sakupljaju u obliku zapisa koji predstavlja proširenje zapisa koji se distribuira uz Kieker.

Novi *Monitoring Log Writer – DProfWriter* – ne upisuje zapise direktno u *Monitoring Log* (kao što je slučaj kod ostalih, *Kieker*-ovih originalnih, *Monitoring Log Writera*), nego u *ResultBuffer*. Bafer šalje zapise na prijemnu stranu. Zapisi mogu da se šalju periodično (upotrebom

JMX *Timer* servisa) ili po zahtevu koji stiže od *Analyzer* komponente (preko *DProfManager* komponente). *RecordReciever* komponenta prihvata zapise i smešta ih u bazu podataka. Praktično, *ResultBuffer*, *RecordReciever* i baza podataka igraju ulogu *Monitoring Log*-a u *Kieker* okruženju. Upotrebom *ResultBuffer* komponente, opterećenje koje DProf sistem izaziva je manje od opterećenja pri upotrebi ostalih Kieker *writer* komponenti. Naime, pošto *ResultBuffer* može da šalje rezultate u paketima, dodatno opterećenje se javlja samo periodično. Sistem je nabolje konfigurisati da pošalje pakete sa rezultatima u periodima kada se očekuje manja aktivnost korisnika sistema.

Komponenta *DProfManager* kontroliše rad bafera i *AspectController*-a – komponente koja kontroliše softverske sonde (*Monitoring Probe*) i konfiguriše rad AOP okruženja. Komponenta *Analyzer* vrši analizu pristiglih zapisa. Na osnovu tih zapisa, definiše novu konfiguraciju i šalje je *DProfManager*-u. Konfiguracija praćenja se definiše preko *aop.xml* datoteke – koja se inače koristi za konfigurisanje AOP okruženja. Svaka izmena konfiguracije praćenja rezultuje izmenama u konfiguracionoj datoteci *aop.xml*.

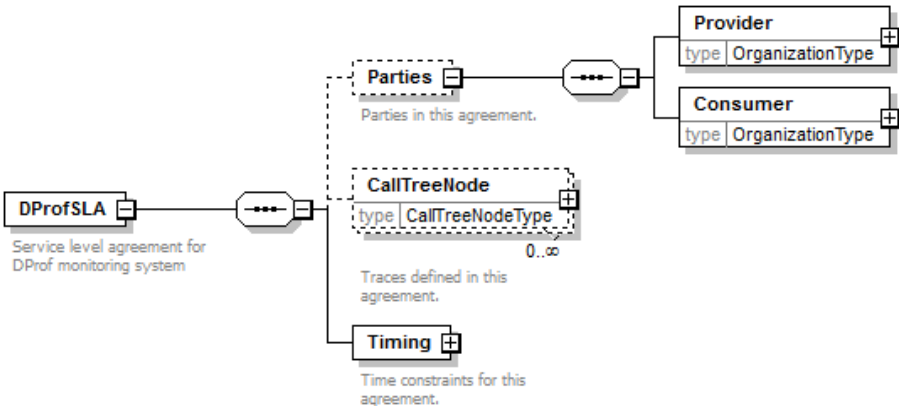
Komunikacija preko mreže se odvija putem veb-servisa. Koriste se JAX-WS veb-servisi. Komponente koje su na istoj strani, komuniciraju kroz virtuelnu mašinu, tj. realizovane su kao JMX *MBean* komponente i komuniciraju kroz *MBeanServer*.

3.2 Definisanje SLA

Sistem koji je prikazan u ovom radu koristi jednostavniju XML shemu za definisanje SLA. Nova shema je definisana, jer za potrebe sistema prikazanog u ovom radu nije potrebna sva kompleksnost shema koje se najviše koriste u ovoj oblasti (npr. GXLA, WSLA). Ova shema predstavlja podskup pomenutih, pa ako je potrebno, dokumenti napisani po ovoj shemi mogu da se prebace u neki drugi standard pomoću XSLT transformacija.

Prema kategorizaciji koja je data u [Paschke06] dokumenti napisani u ovoj shemi spadaju u standardne (prema podeli prema raznovrsnosti primene), operativne ugovore (prema podeli prema nameni), koji se koriste u okviru organizacije (prema podeli prema okviru primene).

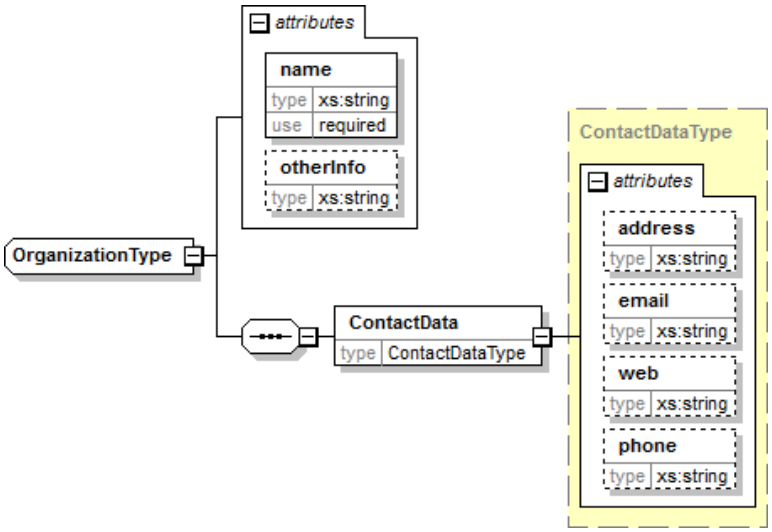
Osnovna struktura sheme prikazana je na slici 3.2.



Slika 3.2 Korenski element sheme za definisanje SLA u DProf sistemu

U elementu *Parties* navode se ugovorne strane, a element *Timing* sadrži vremenske parametre za primenu ugovora. Element *CallTreeNode* sadrži sva stabla poziva koja, po ovom ugovoru, treba da se prate.

Element *Parties* sadrži podatke o ugovornim stranama. Pružalac usluga je predstavljen elementom *Provider*, a korisnik elementom *Consumer*. Oba elementa su predstavljena kompleksnim tipom *OrganizationType* prikazanim na slici 3.3.

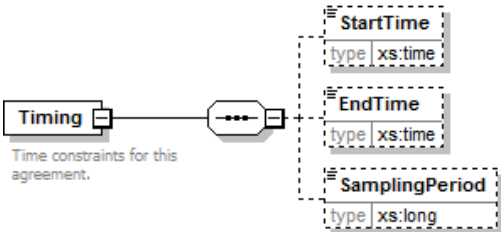


Slika 3.3 Kompleksni tip *OrganizationType*

Kompleksni tip *OrganizationType* sadrži osnovne podatke o organizaciji. Atribut name predstavlja naziv organizacije. Element

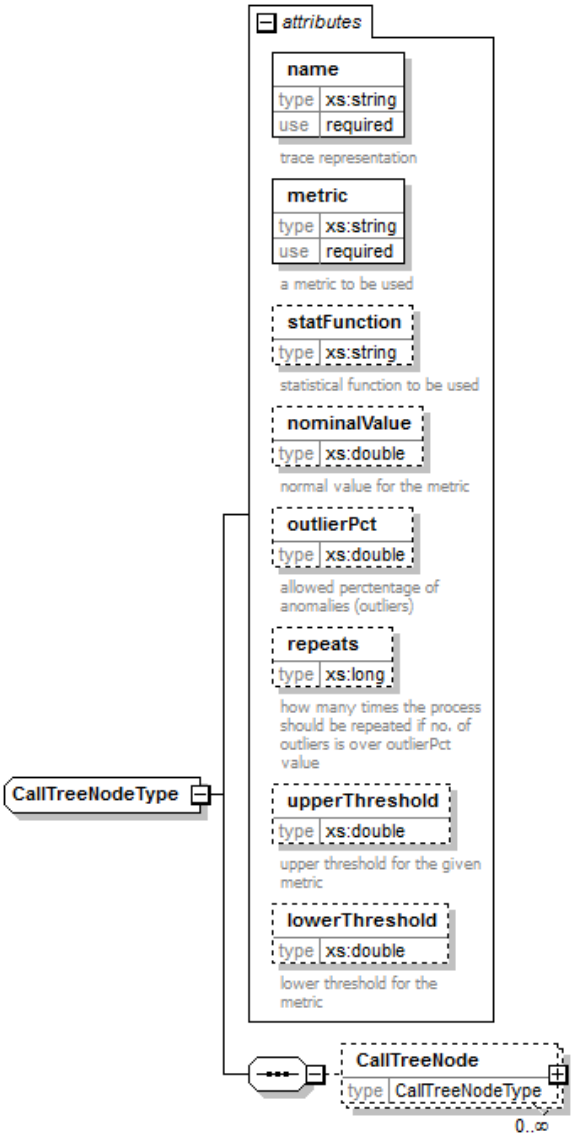
ContactData (predstavljen kompleksnim tipom *ContactDataType*) sadrži kontakt podatke o organizaciji. Atributi definisani u ovom tipu su: *adress* (adresa organizacije), *email* (kontakt e-mail organizacije), *web* (veb adresa organizacije) i *phone* (kontakt telefon organizacije). Svi atributi su predstavljeni prostim tipom *string*.

Element *Timing* (slika 3.4) sadrži tri pod-elementa. *StartTime* i *EndTime* su predstavljeni prostim tipom *time*. Podelement *StartTime* označava vreme kada treba da počne praćenje rada aplikacije. *EndTime* predstavlja vreme kad praćenje treba da se završi. Element *SamplingPeriod* predstavlja vreme između dve rekonfiguracije procesa praćenja. JMX *Timer* okida *Analyzer* komponentu i pokreće se proces analize nakon isteka ovog perioda. Sva tri elementa su prostog tipa *long* i predstavljaju vreme u milisekundama. StartTime i EndTime predstavljaju vreme u milisekundama koje je proteklo od ponoći, 1. januara 1970. (kao u Unix i Java specifikaciji).



Slika 3.4 Element *Timings*

U elementu *CallTreeNode* navode se stabla poziva (pog. 3.4) koja će biti praćena. Elementi *CallTreeNode* su kompleksnog tipa *CallTreeNodeType* (slika 3.5).



Slika 3.5 Element *CallTreeNode*

Elementi kompleksnog tipa *CallTreeNode* sadrže atribut *name* koji odgovara string-reprezentaciji stabla poziva. Atribut *metric* definiše meru koja se koristi u merenjima. Nazivi za mere odgovaraju nazivima koje se koriste u `otherData` atributu klase `DProfExecutionRecord` i

OTHERDATA koloni u tabeli MONITORINGDATA (opisanoj u poglavlju 3.11).

Atribut *statFunction* predstavlja statističku funkciju koja se koristi u obradi podataka (npr. srednja vrednost, standardna devijacija, maksimum, minimum). *outlierPct* je tolerisani procenat vrednosti koje se izbacuju iz posmatranog skupa podataka. Ako je dobijeni procenat veći od zadate vrednosti, merenje treba da se ponovi. Ako se prekorači dozvoljen broj ponavljanja, definisan atributom *repeats*, prijavljuje se problem.

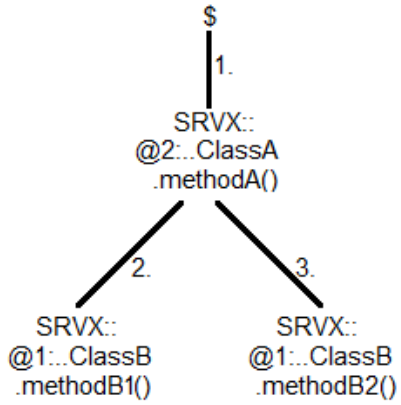
U zavisnosti od izabrane statističke funkcije, koriste se atributi *nominalValue* (nominalna vrednost), *upperThreshold* (gornja granična vrednost) i *lowerThreshold* (donja granična vrednost).

Primer jednog DProfSLA dokumenta dat je na sledećem listingu.

```
<DProfSLA>
 <Parties><Provider name="Org1" />
  <Consumer name="Org2" /></Parties>
 <Trace metric="avgExecutionTime"
   name="[{ClassA.methodA,[{ClassB.methodB1,[]},
   {ClassB.methodB2,[]}]" upperThreshold="350">
 <Trace metric="avgExecutionTime"
   name="[{ClassB.methodB1,[]}]"
   upperThreshold="150"/>
 <Trace metric="avgExecutionTime"
   name="[{ClassB.methodB2,[]}]"
   upperThreshold="150"/>
 <Timing>
   <SamplingPeriod>600000</SamplingPeriod>
 </Timing>
</DProfSLA>
```

Listing 3.1 Primer DProfSLA dokumenta

U ovom dokumentu definiše se ugovor između organizacija Org1 (pruža uslugu) i Org2 (korisnik usluge). Ugovor definiše praćenje metoda `methodA()` iz klase `ClassA` i metoda `methodB1()` i `methodB2()` iz klase `ClassB`. Metoda `methodA()` poziva metode `methodB1()` i `methodB2()`. Grafička reprezentacija stabla poziva, za ovaj slučaj, prikazana je na slici 3.6.



Slika 3.6 Stablo poziva za dati primer

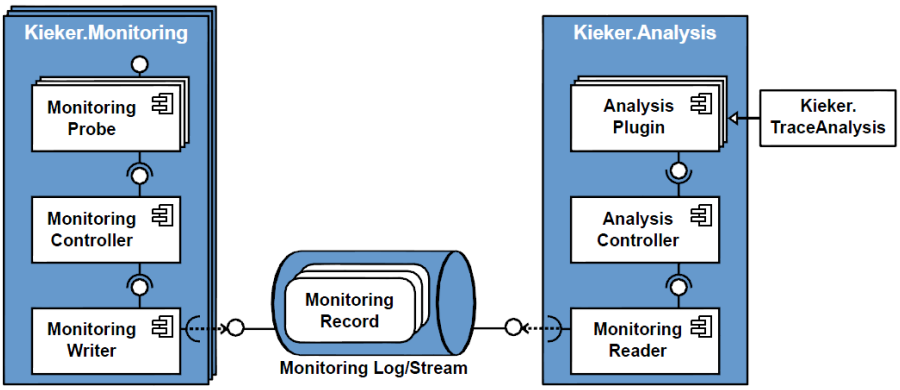
Za sve tri metode definisana su maksimalna vremena odziva, koja ne smeju da se prekorače. Provera dobijenih podataka se vrši na svakih 10 minuta (600000ms). U provm prolazu se prati samo `methodA()`. Ako dođe do prekoračenja, uključuje se praćenje u drugom nivou stabla.

Posle novih 10 minuta, vrši se nova provera podataka. Ako se pojavi prekoračenje u nekoj od metoda u drugom nivou, ona je identifikovana kao izvor problema. Ako metode drugog nivoa ne prekorače zadato vreme odziva, `methodA()` je identifikovan kao problem.

3.3 Kieker okruženje

U [Rohr08] predstavljeno je okruženje Kieker, koje se koristi za kontinuirano praćenje softvera. Okruženje definiše: proširivi model podataka dobijenih profajliranjem (eng. *monitoring record*), proširivi model skladišta (za čuvanje i razmenu podataka između generatora i čitača podataka) i proširivi model obrađivača podataka (podaci mogu da se analiziraju i prikazuju u različite svrhe - dijagrami sekvenci, dijagrami zavisnosti, lanci Markova).

Arhitektura Kieker-a je data na slici 3.7.



Slika 3.7 Arhitektura Kieker okruženja (preuzeto iz [Ehmke12])

Dve osnovne komponente okruženja su Kieker.Monitoring, koja smešta zapise (Monitoring Log Record) u Monitoring Log i Kieker.Analysis, koja obezbeđuje infrastrukturu za analizu i vizualizaciju dobijenih podataka.

Obe komponente okruženja rade nezavisno jedna od druge. Ovo omogućava da, npr. grupa servera (eng. *cluster*) izvršava softver koji se prati. Podaci dobijeni praćenjem mogu da se čuvaju na nekom drugom, a analiziraju na trećem serveru.

Instrumentacija softvera se vrši preko softverskih sondi. Može da se vrši na različite načine, najčešće preko različitih alata za AOP. Referentna implementacija je urađena uz upotrebu AspectJ alata. Presretanje se najčešće vrši pomoću *around advice*-a, tako što se zabeleži stanje sata pre i posle poziva metode. Zapisi koje generišu sonde, prosleđuju se kontroleru (Monitoring Controller), a on ih upisuje u skladište pomoću Monitoring Log Writter-a. AOP okruženje može da se podesi da se presreću samo klase iz određenih paketa ili sve. U aspektima može da se definiše da li da se prate sve metode ili samo one koje su označene određenom anotacijom. Pomoću ove dve tehnike se fino bira koji delovi aplikacije se prate, što dalje omogućava da se bira odnos između željene količine podataka i *overhead*-a.

Čitanje podataka iz skladiša i njihovo pretvaranje u zapise vrši Monitoring Log Reader. Dobijeni zapisi se prosleđuju Monitoring Log Consumer-u, koji se obično pravi tako da prihvata samo zapise određenog tipa. Zadatak Consumer-a je da izvrši analizu ili vizualizaciju komponenti. Kontrolu rada Kieker.Analysis komponente vrši Analysis Controller.

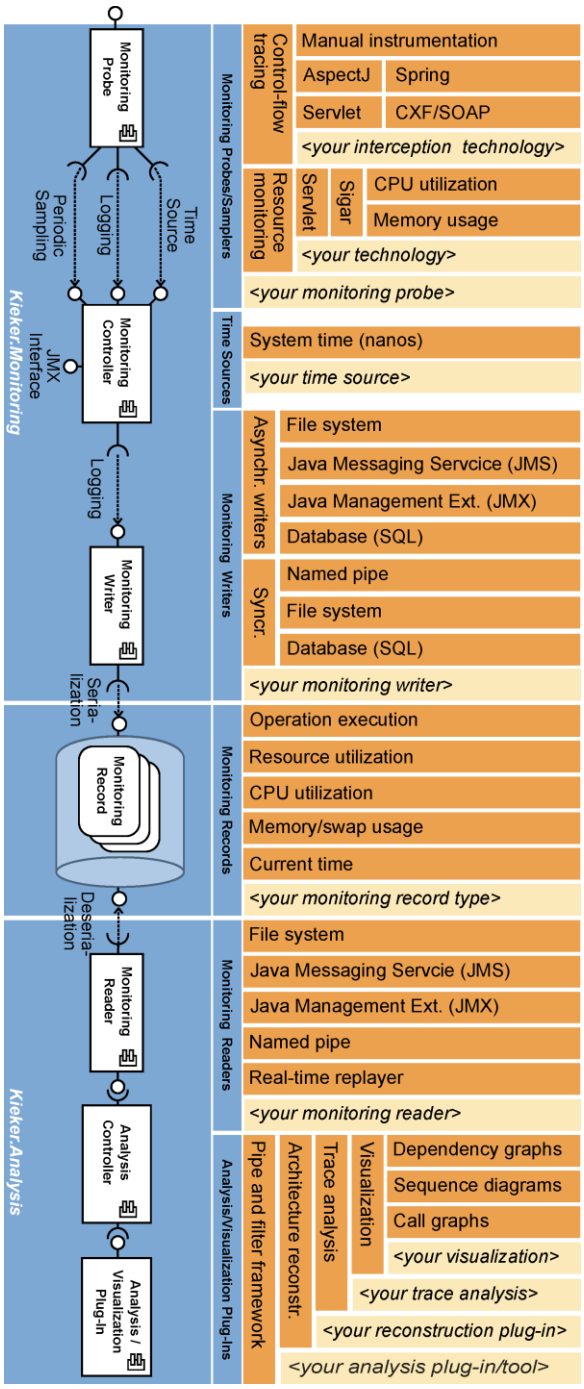
Vizualizacija podataka u Kieker okruženju može da se vrši na više načina [Rohr08]. Kieker.Analysis komponenta, može da generiše UML dijagrame sekvenci, lance Markova (daje prikaz dinamičkog ponašanja sistema pomoću dijagrama stanja; stanja u ovim dijagramima predstavljaju

pozive operacija/kreiranje poruka; prelazak između stanja je okarakterisan verovatnoćom prelaza), dijagrame zavisnosti komponenti (UML dijagram komponenti proširen težinskim faktorima zavisnosti između komponenti, koji označavaju broj poziva komponente) i dijagrame vremenskih rasporeda poziva (predstavlja odnose trajanja izvršavanja u jednoj grani stabla konteksta poziva).

Skladište u kom se čuvaju zapisi može da bude baza podataka, datoteka, JMS red (eng. *queue*) ili nešto drugo. Svaki tip zapisa nasleđuje klasu *AbstractMonitoringRecord*. Referentna implementacija obezbeđuje još i *OperationExecutionMonitoringRecord*, zapis koji sadrži informaciju o izvršenoj operaciji, računaru na kom se izvršava posmatrana aplikacija i vremenske podatke. Vremenski podaci se beleže u rezoluciji nanosekunde, pošto se vreme očitava pozivom *System.nanoTime()*.

Ako je potrebno da se vrši merenje nekih drugih parametara, onda je potrebno da se definiše novi tip softverske sonde (nova aspekt klasa u okviru koje se vrše merenja) i novi tip zapisa (nova klasa koja nasleđuje *AbstractMonitoringRecord*). Za praćenje rada jedne aplikacije mogu da se koriste različiti aspekti i anotacije, tako da je moguće da se prati više različitih parametara istovremeno.

Sve mogućnosti za proširenje Kieker okruženja prikazane su na slici 3.8.



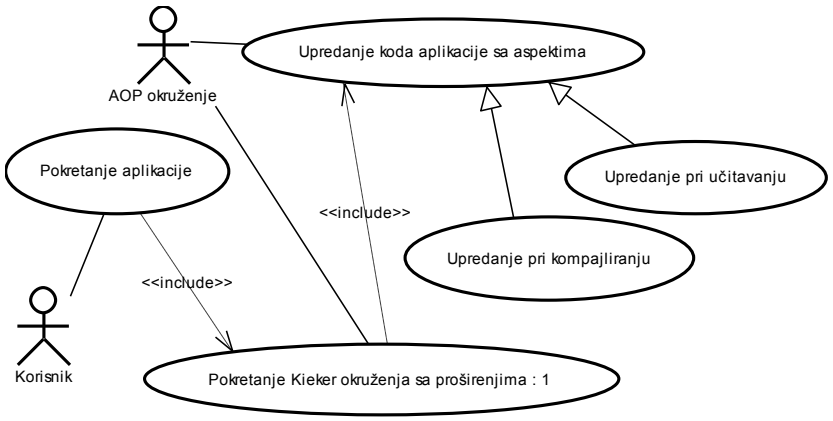
Slika 3.8 Mogućnosti proširenja Kieker okruženja (preuzeto iz [Ehmke12])

3.4 Opis funkcija sistema

Funkcionalnosti u sistemu su prikazane pomoću dijagrama slučajeva korišćenja.

3.4.1 Pokretanje aplikacije i okruženja

Pokretanje aplikacije i Kieker okruženja (sa proširenjima) prikazano je na slici 3.9.



Slika 3.9 Dijagram slučajeva korišćenja za pokretanje aplikacije i DProf okruženja

Kada korisnik pokreće aplikaciju, automatski se pokreće i Kieker okruženje. Koristi se AOP za upredanje aspekata sa klasama aplikacije. Moguće je izvršiti upredanje pri učitavanju i upredanje pri kompajliranju. U ovom sistemu mogu da se koriste oba pristupa. Upredanje pri kompajliranju se koristi ako se ne vrši adaptacija, nego samo praćenje. Klase se upredaju sa aspektima, pokreće se aplikacija i vrši se praćenje.

Adaptacija može da se vrši samo ako se koristi upredanje pri učitavanju, jer se pri adaptaciji vrši uklanjanje starih i učitavanje novih klasa (upredenih sa novim aspektima).

Detaljan opis slučajeva korišćenja sa slike dat je u tabelama.

Naziv	Pokretanje aplikacije
Kratak opis	Korisnik pokreće aplikaciju
Učesnici	Korisnik
Pretpostavke	-
Opis	Korisnik pokreće aplikaciju čiji rad želi da prati. Nije bitno da li je u pitanju samostalna ili distribuirana aplikacija.

Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Pokrenuta aplikacija.

Naziv	Pokretanje Kieker okruženja sa proširenjima
Kratak opis	Pokreće se Kieker okruženje sa proširenjima koja su potrebna za rad DProf sistema.
Učesnici	AOP okruženje,
Pretpostavke	Pokreće se aplikacija.
Opis	Kada se pokrene aplikacija, pokreće se i Kieker okruženje sa proširenjima. AOP vrši povezivanje okruženja i aplikacije.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Pokrenuto okruženje.

Naziv	Upredanje koda aplikacije sa aspektima
Kratak opis	AOP okruženje upreda kod aplikacije sa aspektima
Učesnici	AOP okruženje
Pretpostavke	-
Opis	AOP okruženje vrši upredanje aspekata sa klasama aplikacije i dobijaju se nove klase, koje se koriste umesto starih.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Dobijene su upredene klase koje mogu da se koriste.

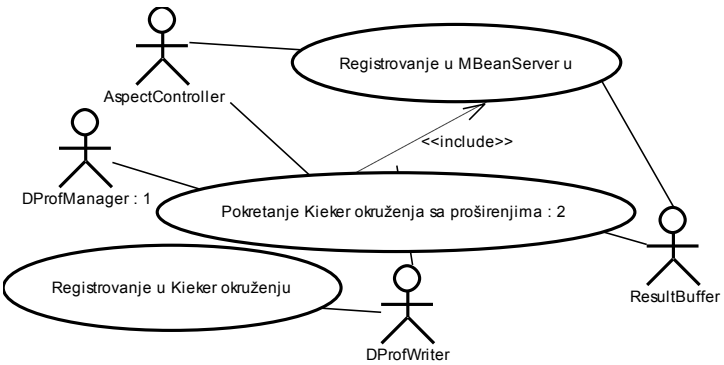
Naziv	Upredanje pri učitavanju
Kratak opis	Upredanje aspekata sa klasama pri učitavanju klase u JVM.
Učesnici	AOP okruženje
Pretpostavke	Pokrenuto Kieker okruženje.
Opis	Kada je potrebno da se klasa učita u JVM, AOP okruženje upreda ovu klasu sa zadatim aspektima.
Izuzeci	-
Uslovi	Dobijene su upredene klase koje se učitavaju u JVM.

zadovoljeni posle izvršavanja	

Naziv	Upredanje pri kompajliranju
Kratak opis	Upredanje aspekata sa klasama pri kompajliranju klase u JVM.
Učesnici	AOP okruženje
Pretpostavke	
Opis	Klasa se upreda sa aspektima pomoću aspekt-kompajlera još u fazi razvoja aplikacije. Dobijena klasa se učitava u JVM kao svaka druga klasa.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Dobijene su upredene klase koje se učitavaju u JVM.

3.4.2 Pokretanje komponenata okruženja

Kada se pokreće Kieker okruženje, pokreću se i dodatne komponente koje predstavljaju proširenje Keker okruženja. Komponente koje se pokreću su DProfWriter, DProfManager, ResultBuffer i AspectController. DProfWriter komponenta mora da se registruje u Kieker okruženju. AspectController i ResultBuffer komponente u napravljene kao MBean komponente, pa moraju da se registruju u MBeanServeru.



Slika 3.10 Dijagram slučajeva korišćenja za pokretanje komponenata DProf okruženja

Detaljan opis slučajeva korišćenja sa slike dat je u tabelama.

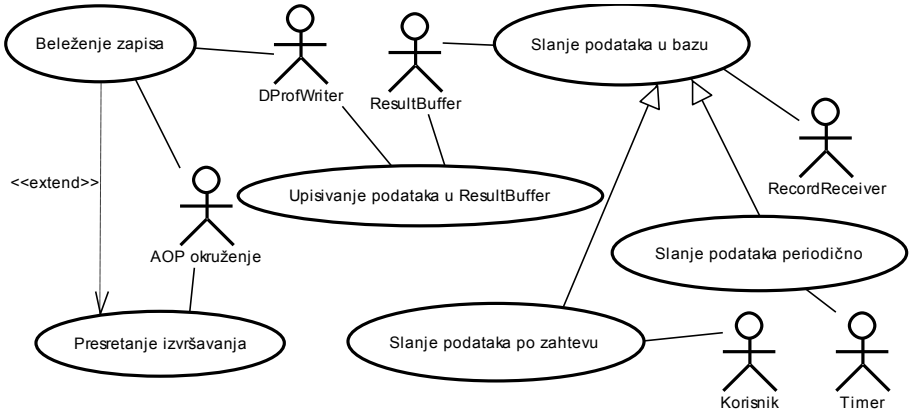
Naziv	Registrovanje u Kieker okruženju

Kratak opis	DProfWriter se registruje u Kieker okruženju.
Učesnici	DProfWriter
Pretpostavke	Pokrenuto Kieker okruženje.
Opis	DProfWriter se registruje u Kieker okruženju i Kieker.Monitoring komponenta koristi njega za beleženje MonitoringRecord-a.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	DProfWriter može da se koristi da upisuje podatke u ResultBuffer.

Naziv	Registrovanje u MBeanServer-u
Kratak opis	Svaka komponenta koja je implementirana kao MBean treba da se registruje u MBeanServer.
Učesnici	AspectController, ResultBuffer
Pretpostavke	Pokrenuto Kieker okruženje.
Opis	AspectController i ResultBuffer su MBean komponente i treba da se registruju u okviru MBeanServer-a da bi ostale komponente mogle da ih koriste.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	AspectController i ResultBuffer mogu da se koriste.

3.4.3 Beleženje i slanje podataka

AOP okruženje presreće izvršavanje i, ako je izvršavanje pokriveno nekim od *joinpoint*-a, podatak o izvršavanju se šalje *DProfWriter*-u. *DProfWriter* upisuje podatak u *ResultBuffer*. *ResultBuffer* šalje podatke periodično (okidanje slanja vrši JMX *Timer* servis) ili po zahtevu koji šalje korisnik.



Slika 3.11 Dijagram slučajeva korišćenja za proces beleženja događaja i slanja zapisa

Detaljan opis slučajeva korišćenja sa slike dat je u tabelama.

Naziv	Presretanje izvršavanja
Kratak opis	AOP okruženje presreće poziv metode.
Učesnici	AOP okruženje
Pretpostavke	Pokrenuta aplikacija.
Opis	AOP okruženje detektuje izvršavanje određene metode. Ovo se vrši pomoću koda koji je upreden u aspekte.
Izuzeci	Pozvano izvršavanje nije definisano u aspektima ili *aop.xml*.
Uslovi zadovoljeni posle izvršavanja	-

Naziv	Beleženje zapisa
Kratak opis	Poziva se *DProfWriter* da se zabeleži zapis.
Učesnici	AOP okruženje, *DProfWriter*, Kieker okruženje
Pretpostavke	Pokrenuto Kieker okruženje.
Opis	AOP okruženje poziva *DProfWriter* da upiše zapis u *Monitoring Log*. Ovo se sve vrši kroz Kieker okruženje.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Zapis je prosleđen DProfWriter-u.

Naziv	Upisivanje podataka u *ResultBuffer*
Kratak opis	*DProfWriter* upisuje zapis u *ResultBuffer*.
Učesnici	*DProfWriter*, *ResultBuffer*
Pretpostavke	Pokrenuto Kieker okruženje.
Opis	*DProfWriter*, kroz *MBeanServer*, poziva *ResultBuffer* i prosleđuje mu zapis.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Zapis je u *ResultBuffer*-u.

Naziv	Slanje podataka *RecordReceiver*-u
Kratak opis	*ResultBuffer* šalje podatke *RecordReceiver*-u.
Učesnici	*ResultBuffer*, *RecordReceiver*
Pretpostavke	Pokrenuto Kieker okruženje. Pokrenut *RecordReceiver*.
Opis	*ResultBuffer* zapise pretvara u tekst i šalje ih *RecordReceiver*-u putem veb-servisa.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Zapis je prosleđen *DProfWriter*-u.

Naziv	Slanje podataka po zahtevu
Kratak opis	*ResultBuffer* šalje podatke *RecordReceiver*-u na zahtev korisnika.
Učesnici	*ResultBuffer*, *RecordReceiver*, Korisnik
Pretpostavke	Pokrenuto Kieker okruženje. Pokrenut *RecordReceiver*.
Opis	*ResultBuffer* šalje podatke *RecordReceiver*-u na zahtev korisnika. Korisnik, putem JMX konzole, izdaje *ResultBuffer*-u komandu da pošalje zapise.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Zapis je prosleđen *RecordReceiver*-u.

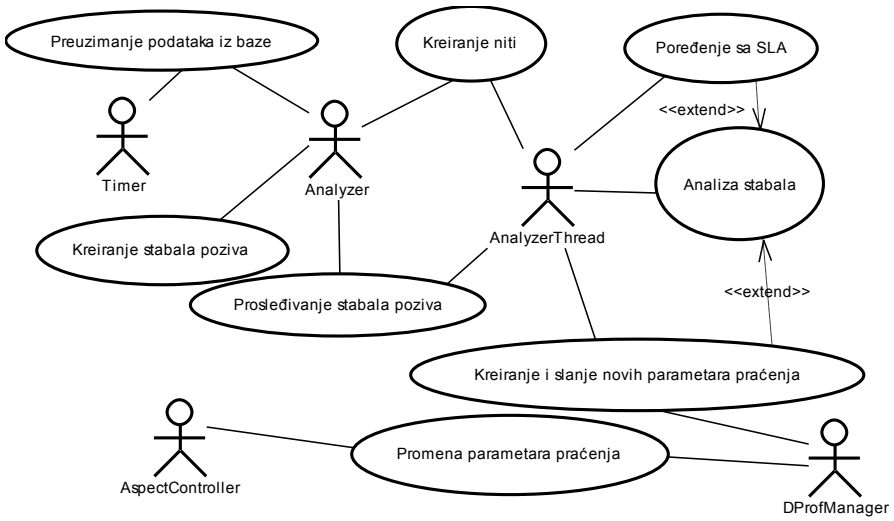
Naziv	Slanje podataka periodično
Kratak opis	*ResultBuffer* šalje periodično podatke *RecordReceiver*-u.
Učesnici	*ResultBuffer*, *RecordReceiver*, *Timer* servis
Pretpostavke	Pokrenuto Kieker okruženje. Pokrenut

	RecordReceiver. Pokrenut *Timer* servis.
Opis	JMX *Timer* servis šalje notifikacije u pravilnim intervalima. Po prijemu notifikacije, *ResultBuffer* šalje prikupljene zapise *RecordReceiver*-u putem veb-servisa.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Zapis je prosleđen *DProfWriter*-u.

3.4.4 Analiza podataka

JMX *Timer* servis periodično šalje notifikacije *Analyzer*-u. *Analyzer* kreira stabla poziva i niti (*AnalyzerThread*). Stabla se prosleđuju pojedinačnim nitima. Koriste se niti, da bi se proces analize paralelizovao. Niti vrše poređenje dobijenih rezultata u stablima sa vrednostima definisanim u SLA, ako je potrebno. Ako nije, *Analyzer* vrši snimanje stabala. Na osnovu analize, ako je potrebno, *AnalyzerThread* kreira i šalje nove parametre *DProfManager*-u. *DProfManager* prima ove parametre i vrši rekonfiguraciju okruženja prosleđujući nove parametre *AspectController* komponenti. Dalje praćenje se vrši sa novim parametrima.

Detaljan opis slučajeva korišćenja sa slike dat je u tabelama.



Slika 3.12 Dijagram slučajeva korišćenja za proces analize podataka

Naziv	Preuzimanje podataka iz baze
Kratak opis	Analyzer periodično preuzima podatke za analizu.

Učesnici	*Timer*, *Analyzer*
Pretpostavke	Pokrenut *Timer* servis. *Analyzer* prijavljen za notifikacije.
Opis	Kada *Analyzer* dobije notifikaciju od *Timer*-a, on preuzima podatke iz baze.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	*Analyzer* je preuzeo sve podatke i spreman je da kreira stabla poziva.

Naziv	Kreiranje stabala poziva.
Kratak opis	*Analyzer* kreira stabla poziva.
Učesnici	*Analyzer*
Pretpostavke	Učitani podaci iz baze.
Opis	*Analyzer* kreira stabla poziva na osnovu podataka preuzetih iz baze podataka.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Kreirana stabla poziva sa vremenima izvršavanja.

Naziv	Kreiranje niti
Kratak opis	*Analyzer* kreira *AnalyzerThread* programske niti.
Učesnici	*Analyzer*, *AnalyzerThread*
Pretpostavke	Kreirana stabla poziva.
Opis	Za svako stablo poziva *Analyzer* kreira *AnalyzerThread* programsku nit.
Izuzeci	Već postoji nit za to stablo poziva.
Uslovi zadovoljeni posle izvršavanja	Kreirane *AnalyzerThread* programske niti.

Naziv	Prosleđivanje stabala poziva
Kratak opis	*Analyzer* prosleđuje dobijeno stabla poziva odgovarajućoj niti.
Učesnici	*Analyzer*, *AnalyzerThread*
Pretpostavke	Kreirana stabla poziva. Kreirane programske niti.
Opis	*Analyzer* prosleđuje svako stablo poziva odgovarajućoj programskoj niti.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Niti su spremne za pokretanje.

Naziv	Analiza stabala
Kratak opis	*AnalyzerThread* se pokreće i poziva R okruženje. Vrši se analiza dobijenih rezultata.
Učesnici	*AnalyzerThread*
Pretpostavke	Kreirana stabla poziva. Kreirane programske niti.
Opis	*AnalyzerThread* se pokreće i poziva R okruženje. R okruženje će da izračuna prosečnu vrednost (u odgovarajućoj metrici) i broj ekstremnih vrednosti. Ako je potrebno, vrši se poređenje sa SLA (slučaj korišćenja "Poređenje sa SLA"). Dobijeni rezultati (broj ekstremnih vrednosti, prosečne vrednosti) se analiziraju. Ako je broj ekstremnih vrednosti prevelik, merenje treba da se ponovi (pošto je verovatno bilo dosta ometanja u procesu merenja). Ako nema poređenja sa SLA, onda se treba da se vrši dalje praćenje.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Dobijen broj ekstremnih vrednosti i prosečne vrednosti.

Naziv	Poređenje sa SLA
Kratak opis	*AnalyzerThread* poredi dobijene rezultate iz analize sa zadatim SLA.
Učesnici	*AnalyzerThread*
Pretpostavke	Definisan SLA. Dobijen broj ekstremnih vrednosti i prosečne vrednosti.
Opis	*AnalyzerThread* poredi dobijene rezultate iz analize sa vrednostima zadatim u SLA.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Vrednosti dobijene merenjem su upoređene sa SLA i na osnovu toga može da se formira nova konfiguracija praćenja.

Naziv	Kreiranje i slanje novih parametara praćenja
Kratak opis	*AnalyzerThread* šalje *DProfManager*-u nove parametre praćenja.
Učesnici	*AnalyzerThread*, *DProfManager*
Pretpostavke	Izvršena analiza stabala.

Opis	Ako je analiza pokazala da treba da se izvrši izmena, *AnalyzerThread* šalje *DProfManager*-u nove parametre praćenja.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Novi parametri praćenja poslati *DProfManager*-u.

Naziv	Promena parametara praćenja
Kratak opis	*DProfManager* vrši izmenu parametara praćenja u okruženju.
Učesnici	*DProfManager*
Pretpostavke	Novi parametri koje je definisao *AnalyzerThread* su stigli *DProfManager*-u.
Opis	*DProfManager* vrši izmenu parametara praćenja u okruženju. On šalje odgovarajuće komande *AspectController*-u.
Izuzeci	-
Uslovi zadovoljeni posle izvršavanja	Okruženje nastavlja praćenje sa novim parametrima.

3.5 Proces praćenja pomoću DProf sistema

Globalni dijagram aktivnosti koji prikazuje celokupan proces praćenja dat je na slici 3.13.

Pre nego što se pokrene aplikacija, inicijalna konfiguracija se zadaje anotiranjem metoda koje želimo da pratimo i/ili pomoću include/exclude stavki u aop.xml konfiguracionoj datoteci AspectJ okruženja.

Ceo proces počinje pokretanjem aplikacije. Pri pokretanju aplikacije, pokreću se i okruženje i *Analyzer* komponenta. Proces se grana na prikupljanje i analizu podataka. Kieker okruženje sa DProf proširenjem vrši prikupljanje podataka u toku izvršavanja aplikacije.

Prikupljanje podataka prikazano je dijagramom aktivnosti datim na slici 3.14.

Tokom izvršavanja aplikacije, Kieker okruženje presreće pozive u izvršavanju aplikacije i vrši potrebna merenja. Dobijene vrednosti se šalju *ResultBuffer* komponenti. *ResultBuffer* dalje prosleđuje podatke u *RecordReceiver*-u.

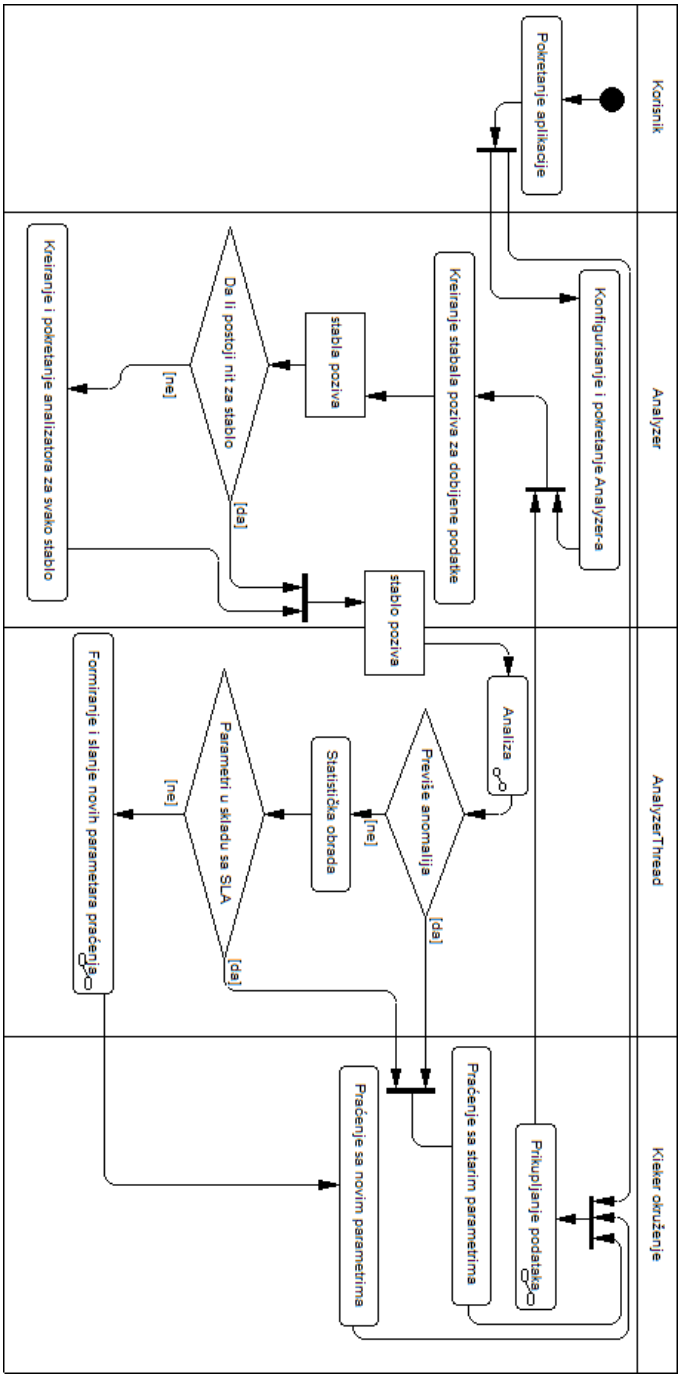
Analyzer komponenta periodično uzima prikupljene podatke iz baze podataka i od njih kreira stabla poziva. Za svako stablo predviđeno SLA ugovorom se kreira programska nit koja vrši analizu dobijenih stabala.

Moguće je da nit za takvo stablo već postoji, pa se kreiranje i u tom slučaju preskače, nego se toj već postojećoj niti prosleđuje odgovarajuće stablo.

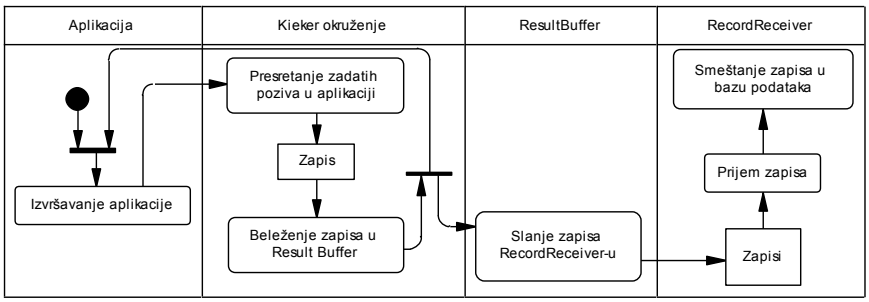
Niti vrše statističku analizu podataka. Prvo se broj anomalija poredi sa brojem zadatim u SLA dokumentu. Ako je u podacima pronađeno previše anomalija, proces praćenja za to stablo se ponavlja sa istim parametrima. Ovo se radi da bi se dobila realna slika rezultata. Ako se i nakon ponavljanja dobije prevelik broj anomalija, smatra se da u sistemu postoji problem.

Ako je broj anomalija prihvatljiv, proverava se da li su parametri u skladu sa SLA. Ako jesu praćenje se nastavlja sa parametrima koji obuhvataju samo koren stabla. Ako parametri nisu u skladu sa SLA, proces se nastavlja dalje sa novim parametrima koji će obuhvatiti nove nivoe u stablu poziva.

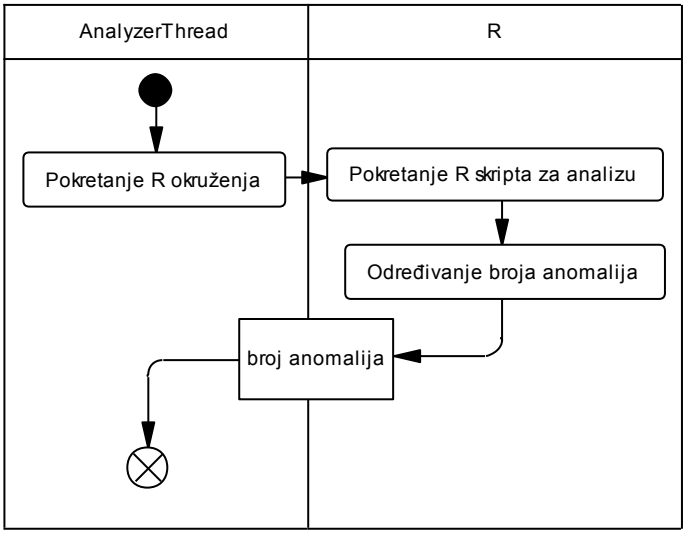
Dijagram aktivnosti za proces analize dat je na slici 3.15.



Slika 3.13 Dijagram aktivnosti koji opisuje rad DProf sistema



Slika 3.14 Dijagram aktivnosti za proces praćenja i beleženja zapisa



Slika 3.15 Dijagram aktivnosti za proces analize podataka

Programske niti koriste R programski jezik [RLanguage10] i dodatni paket *extremevalues* [Loo11], za statističku analizu dobijenih rezultata.

R je standardni programski jezik za statističku obradu i vizualizaciju podataka. Razvijen je pod GNU licencom. Koristi se za razvoj statističkog softvera i analizu podataka. R podržava različite statističke i grafičke tehnike koje uključuju linearno i nelinearno modelovanje, klasične statističke testove, analizu vremenskih serija, klasifikaciju i klasterovanje. R koristi komandnu liniju kao za komunikaciju sa korisnikom, ali je dostupno i nekoliko grafičkih okruženja.

R se proširuje kroz dodatne pakete. Paketi mogu da se razvijaju u različitim jezicima (R, Java, C, …).

Paket *extremevalues* sadrži funkcije za detekciju anomalija (eng. *outlier detection*). U statistici se za anomalije smatraju pojave koje su numerički drastično udaljene od ostatka skupa podataka. Najčešći uzrok pojave anomalija su greške merenja. Paket extremevalues sadrži funkcije koje procenjuju koje vrednosti su anomalije.

Za zadati skup podataka, regresijom se utvrđuju parametri raspodele kojoj pripada skup posmatranih vrednosti. Prema prvom metodu, vrednosti iz skupa podataka koje su izvan zadatog intervala, a u odnosu na raspodelu, su ekstremne vrednosti. Drugi metod uzima odstupanja reziduala posmatranih vrednosti od zadatih vrednosti. Ako je odstupanje veće od zadate granice, vrednost je anomalija.

Paketi *rJava* i *JRI* se koriste za povezivanje R i Java okruženja [Urbanek11]. Pomoću funkcija definisanih u rJava, moguće je izvršavanje Java programa u okviru R okruženja, a pomoću JRI iz Java programa mogu da se pozivaju funkcije napisane u R jeziku.

Pomoću R okruženja prvo se određuje broj anomalija. Ove anomalije su obično posledica različitih spoljašnjih uticaja: učitavanja klasa, pokretanje pozadinskih procesa dok radi aplikacija ili hardverskih poremećaja.

Nakon što se uklone anomalije iz skupa posmatranih vrednosti, vrši se proračun u skladu sa statističkom operacijom zadatom u SLA.

U zavisnosti od rezultata statističke obrade, moguća su tri scenarija:

1. Ako statistička obrada pokaže da su rezultati u skladu sa SLA, praćenje se nastavlja sa starim parametrima.
2. Ako je broj anomalija prevelik, merenje se ponavlja sa starim parametrima, pošto se smatra da dobijeni rezultati ne daju pravu sliku sistema. Ako se ova situacija ponovi još jednom, onda se smatra da postoji problem.
3. Ako statistička obrada da rezultate koji nisu u skladu sa SLA, kreiraju se novi parametri za proces praćenja: utvrđuje se da je u trenutnom čvoru greška ili se uključuje praćenje u sledećem nivou stabla poziva.

3.6 Instrumentacija

Izbor i raspored sondi zavisi od cilja prikupljanja podataka. U praksi se sonde ubacuju *ad-hoc*, najčešće da se ispita razlog greške, neočekivanog prekida rada ili ispitivanja nekog drugog nepredviđenog ponašanja.

Broj sondi koje će biti postavljene mora biti pažljivo izabran jer može direktno da utiče na kvalitet rezultata. Velik broj sondi donosi visok kvalitet rezultata, ali unosi veliko kašnjenje u sistem. S druge strane, mali

broj sondi ne zauzima veliku količinu resursa, ali često ne daje dovoljno precizne rezultate.

Najjednostavniji način da se izvrši instrumentacija je da se u sam kod aplikacije umetnu instrukcije koje vrše potrebna očitavanja i smeštaju dobijene podatke u odgovarajuće skladište (*Monitoring Log* u ovom slučaju).

Recimo da treba da se prati opterećenje sistema tokom izvršavanja metode `doSomething()` u aplikaciji na listingu 3.2.

```
1 // ...
2 doSomething();
3 // ...
```

Listing 3.2 Kod koji treba da se prati

Primer za očitavanje opterećenja sistema tokom izvršavanja programskog koda (u programskom jeziku Java) dat je na listingu 3.3 (detaljnije objašnjenje za instrukcije za očitavanje iz ovog primera je u pog. 3.7.2).

```
1 OperatingSystemMXBean opSysMXbean =
    ManagementFactory.getOperatingSystemMXBean();
2 // ...
3 doSomething();
4 // ...
5 System.out.println("Prosecno opterecenje " +
    opSysMXbean.getSystemLoadAverage());
```

Listing 3.3 Praćenje opterećenja sistema tokom izvršavanja jedne metode u Java programa

Merenje prosečnog opterećenja se vrši pomoću `OperatingSystemMXBean` komponente. U listingu se vidi da se prvo pribavi referenca na komponentu, zatim se izvrši merenje i dobijena vrednost se ispiše na ekranu (zbog pojednostavljivanja, nije izvršeno smeštanje u neko posebno skladište).

Prikazan način instrumentacije je loš zato što se u programski kod aplikacije ubacuje kod koji vrši merenje. Na ovaj način dolazi do "zagađivanja" koda aplikacije, pa je mnogo bolje da se koristi AOP.

Ako se koristi AOP, programski kod koji vrši merenja se izdvaja u aspekt. U aspektu se definišu delovi koda koji će biti praćeni (kao *pointcut*-i u aspektu) i način na koji se vrši praćenje (kao *advice* u aspektu).

Programski kod koji treba da se prati može da se prati pomoću sledećeg aspekta prikazanog na listingu 3.4.

```
1 public aspect {
2 pointcut execSomething() : execution(public void
     *.doSomething(...));
3 void around() : execSomething() {
4 OperatingSystemMXBean opSysMXbean =
     ManagementFactory.
  getOperatingSystemMXBean();
5 proceed();
6 System.out.println("Prosecno opterecenje " +
     opSysMXbean.getSystemLoadAverage());
7 }
8 }
```

Listing 3.4 Aspekt za instrumentaciju koda sa listinga 3.2

Pointcut `execSomething()` presreće izvršavanje metode `doSomething()`. U *advice*-u `around()` se definiše praćenje. Operacije pomoću kojih se vrši praćenje sa listinga 3.1 su ubačene u ovaj *advice*. Praktično, sve što je navedeno u *advice*-u izvršava se umesto presretnute metode. Sama metoda `doSomething()` se u okviru *advice*-a poziva pomoću `proceed()` metode.

Postupak primene aspekata na programski kod aplikacije se naziva upredanje (eng. *weaving*). Postoje dve vrste upredanja: upredanje pri kompajliranju (eng. *compile time weaving*) i upredanje pri učitavanju klase u JVM (eng. *load time weaving*).

Ubacivanje softverskih sondi i upredanje aspekata u toku razvoja, se vrši pokretanjem *ajc* alata (skr. *AspectJ Compiler*) i naziva se *compile time weaving*. Kompajler uzima izvorni kod aplikacije i aspekata i generiše nove (upredene) klase.

U slučaju upredanja pri učitavanju, uz aplikaciju se distribuiraju prekompajlirani aspekti. Kada se određena klasa učitava u JVM, AOP okruženje presretne učitavanje, izvrši upredanje aspekta sa tom klasom i u JVM se učitava nova, upredena klasa. Nedostatak ovog pristupa je što pokretanje aplikacije traje malo duže u odnosu na upredanje pri kompajliranju. Prednost upredanja pri učitavanju je što može da se prati rad programa čiji izvorni kod nije dostupan. Konfigurisanje se vrši pomoću parametra komandne linije (pomoću kog se uključuje upredanje) pri pokretanju Java aplikacije i pomoću *aop.xml* konfiguracione datoteke. U okviru *aop.xml* datoteke definišu se parametri za proces upredanja i aspekti

koji se koriste, a moguće je uključiti i isključiti deo programa koji se upreda (selekcija može da se vrši na nivou klasa i paketa).

U okviru ovog sistema, koristi se upredanje pri učitavanju. Osnovni razlog je činjenica da programski kod aplikacije nije uvek dostupan i što je u tom slučaju lakše uključivanje i isključivanje praćenja. Okruženje može da kreira *aop.xml* i da ga smesti u CLASSPATH aplikacije. Problem sporijeg rada aplikacije u trenucima kada se vrši učitavanje i upredanje sa stanovišta korisnika ne mora da bude toliko drastičan, jer se ovo dešava samo na početku – aplikacija se kasnije izvršava normalnom brzinom. Rezultati koji se dobijaju u trenucima kada se vrši učitavanje nove klase ne daju pravu sliku pa se izbacuju u procesu analize.

Ako je ipak vreme izvršavanja kritično i mora da se vrši upredanje pri kompajliranju, prikupljanje podataka i dalje može da se vrši, ali sa prekonfigurisanim aspektima, bez mogućnosti izmene konfiguracije i parametara praćenja.

3.7 Merenja u Java okruženju

Za merenje zadatih parametara koriste se standardni mehanizmi dostupni u Java platformi. Posebno se razmatra merenje vremena (pog. 3.7.1), posebno merenje pomoću platformskih komponenti (pog. 3.7.2), a posebno merenje brzine protoka (pog. 3.7.3).

3.7.1 Merenje vremena

Uobičajen način za merenje vremena u Java platformi je očitavanjem sistemskog sata. Koristi se klasa `System` i njena metoda `currentTimeMilis()`. Ona kao rezultat vraća broj milisekundi koji je protekao od ponoći, 1.1.1970. Primer upotrebe ove metode dat je na listingu 3.5.

```
1 long startTime = System. currentTimeMillis ();
2 // ...programski kod cije izvrsavanje pratimo...
3 long estimatedTime = System.currentTimeMillis() -
    startTime;
```

Listing 3.5 Primer koda kojim se meri trajanje izvršavanja dela programskog koda

Na kraju, u promenljivoj `estimatedTime`, je smešteno trajanje izvršavanja posmatranog programskog koda.

Od verzije Java platforme 5.0, u `System` klasu je dodata metoda `nanoTime()`. Ova metoda vraća broj nanosekundi od proizvoljnog (nepoznatog) trenutka u vremenu. Trenutak može da bude i u budućnosti,

tako da se dobiju negativne vrednosti. Pošto se meri samo trajanje, a na nivou nanosekunde, ova metoda je pogodnija od `currentTimeMillis()` za merenje trajanja izvršavanja programskog koda.

Preciznost sa kojom se vrše očitavanja je predmet brojnih diskusija i zavisi i od računara na kom se vrši merenje, od operativnog sistema, kao i od implementacije JVM ([Lambert08], [Subotic05]). S druge strane, trajanje izvršavanja nekog dela programskog koda zavisi od više spoljašnjih faktora. Npr. učitavanje klasa ima ozbiljan uticaj na rezultate na početku izvršavanja; aktiviranje nekog drugog procesa u računaru (npr. pokretanje druge aplikacije) izaziva preključivanje procesora na taj proces. Svi ovi faktori mogu da utiču na krajnje rezultate, jer može da dođe do sporijeg odziva.

3.7.2 Merenje pomoću platformskih MXBean komponenti

U okviru Java platforme, od verzije 5.0, dostupno je nekoliko platformskih *MXBean*-ova pomoću kojih možemo da imamo uvid u različite parametre u JVM. Platformski *MXBean* je komponenta pomoću koje može da se vrši praćenje i upravljanje radom JVM i drugih komponenti u okviru Java okruženja u toku rada programa. Svaka od ovih *MXBean* komponenti enkapsulira deo funkcionalnosti u okviru JVM.

Spisak svih platformskih *MXBean* komponenti (po abecednom redu) dat je u tabeli **Tabela 3.1**.

Interfejs	Funkcionalnost u JVM	Objektni naziv	Broj instanci u JVM
`ClassLoadingMXBean`	Učitavanje klasa	`java.lang:type = ClassLoading`	1
`CompilationMXBean`	Sistem za kompajliranje	`java.lang:type = Compilation`	0 ili 1
`GarbageCollector-MXBean`	Garbage collector	`java.lang:type = GarbageCollector, name=`*`collectorName`*	1 ili više
`LoggingMXBean`	Sistem za evidenciju događaja	`java.util.logging:type = Logging`	1
`MemoryManager-MXBean`	Upravljanje memorijom	`java.lang: typeMemoryManager, name=`*`managerName`*	1 ili više

`MemoryPoolMXBean`	Upravljanje memorijskim pool-om	`java.lang: type = MemoryPool, name=`*`poolName`*	1 ili više
`MemoryMXBean`	Memorija	`java.lang:type = Memory`	1
`OperatingSystem-MXBean`	Operativni sistem na kom je JVM	`java.lang:type = OperatingSystem`	1
`RuntimeMXBean`	Runtime sistem	`java.lang:type = Runtime`	1
`ThreadMXBean`	Upravljanje thread-ovima	`java.lang:type = Threading`	1

Tabela 3.1 Platformske *MXBean* komponente

Pomoću `ClassLoadingMXBean` komponente mogu da se očitaju podaci o učitavanju klasa u JVM: broj učitanih klasa (u trenutku očitavanja i ukupan broj klasa koje su učitavane od početka rada JVM) i broj klasa koje su uklonjenje do početka rada.

Ukupno vreme rada *just-in-time* kompajlera može da se očita pomoću `CompilationMXBean` komponente.

Broj poziva *Garbage Collector*-a i ukupno vreme provedeno u sakupljanju objekata koji više nisu u upotrebi se očitava pomoću `GarbageCollectorMXBean`.

Očitavanje parametara sistema za evidentiranje događaja (eng. *logging*) i upravljanje sistemom se vrši pomoću `LoggingMXBean` komponente. Moguće je listanje registrovanih *logger*-a, provera filtera nivoa događaja i rekonfigurisanje sistema (izmena nivoa).

MXBean komponente koje se koriste za proveru zauzetosti memorije su `MemoryMXBean`, `MemoryManagerMXBean` i `MemoryPoolMXBean`. `MemoryMXBean` je MXBean kojim se očitava stanje sistemske memorije. Može da se očitava i stanje dinamičke (eng. *heap*) i ostale memorije. Stanje obe memorije je predstavljeno `MemoryUsage` klasom.

Klasa `MemoryUsage` ima četiri atributa: `init` (inicijalnu količinu memorije koju JVM traži pri pokretanju; ako je potrebno, JVM može kasnije da traži dodatnu memoriju i da oslobodi nepotrebnu memoriju), `used` (količina memorije koja se trenutno koristi), `commited` (količina memorije koja je JVM garantovano dostupna; menja se tokom vremena i uvek je veća od `used`) i `max` (maksimalna količina memorije koja može da

se iskoristi; može da se promeni tokom vremena i uvek je veća od `commited`). Sva četiri atributa su tipa `long` i predstavljaju količinu memorije u bajtovima.

`MemoryMXBean` može da emituje notifikacije u zavisnosti od definisanih pragova zauzetosti memorije, koje drugi *MBean*-ovi u JVM mogu da primaju i obrađuju.

Upravljanje memorijskim *pool*-ovima u okviru JVM vrše *MemoryManager* komponente, čije stanje se nadgleda `MemoryManagerMXBean` komponentama, a svaki *pool* ima svoj `MemoryPoolMXBean`. *Pool* ima četiri atributa: zauzetost memorije (tj. *pool*-a koji se posmatra), maksimalna zauzetost (maksimalna zauzetost od pokretanja JVM), prag zauzetosti (provera da li je zauzetost u *pool*-u veća od zadatog praga; može da se vrši redovnom proverom ili da *pool* pošalje notifikaciju kad je pun) i prag zauzetosti sakupljanja (samo za *pool* koji koristi *Garbage Collector*; provera se vrši nakon pokretanja *Garbage Collector*-a).

Detalji o samoj JVM se očitavaju pomoću `RuntimeMXBean` komponente. Moguće je očitavanje sistemskih promenljivih, Java verzije, vremena kad je startovana JVM, detalja o proizvođaču i verziji JVM, detalja o proizvođaču i verziji JVM specifikacije.

Pomoću `OperationSystemMXBean` se očitavaju podaci o platformi na kojoj se izvršava JVM. Mogu da se očitaju naziv operativnog sistema, arhitekture, verzija, broj procesora dostupnih JVM i prosečno opterećenje sistema. Prosečno opterećenje se odnosi na poslednji minut i njegovo izračunavanje zavisi od platforme na kojoj se izvršava JVM.

Upotreba ovih komponenti daje precizne informacije o radu JVM i aplikacijama koji su u njoj. Tačnija merenja mogla bi, u nekim slučajevima, da se dobiju jedino pristupanjem platformi na kojoj se izvršava JVM, što narušava portabilnost merenja. Takođe, JNI koji mora da se koristi u ovom slučaju može da donese mnogo veću grešku merenja nego što se dobija pomoću platformskih MXBean komponenti ([Binder06], [Rohr08]).

3.7.3 Merenje brzine protoka

Što se tiče mrežnog protoka i nadgledanja mreže, u standardnoj distribuciji, Java nema gotov mehanizam pomoću kog može da se vrše ova merenja. Ovo nije slučaj samo sa mrežnom komunikacijom, nego i sa komunikacijom sa bilo kojim spoljašnjim priključkom (npr. USB, serijski i paralelni). U ovom slučaju moguća su dva pristupa: pristupanje mehanizmima operativnog sistema na kom se izvršava JVM ili merenje vremena potrebnog da se prenese određena količina podataka.

Prvi pristup (upotreba sistemskih mehanizama) podrazumeva upotrebu alata operativnog sistema. Većina operativnih sistema ima mogućnost merenja različitih sistemskih parametara, pa i mrežnog protoka. Problem sa ovim pristupom je što, da bi se vršila ova merenja, neophodno je prepoznati koji je operativni sistem u pitanju, proveriti da li je merenje dozvoljeno i tek onda izvršiti merenje. Zbog svega navedenog, ovaj pristup smanjuje portabilnost procesa merenja.

Pre merenja vremena koje je potrebno da se prenese određena količina podataka, prvo mora da se izračuna količina podataka u bajtovima. Nakon toga se izvrši slanje i izmeri koliko je vremena bilo potrebno. Ako se koristi ova metoda, brzina protoka je:

$$Protok\left[\frac{byte}{s}\right] = \frac{Q\,[byte]}{T_k - T_p}$$

gde je protok (izražen u bajtovima u sekundi) jednak količniku količine podataka koja je preneta (Q – u bajtovima) i vremena koje je potrebno da se prenesu ti podaci (razlika između T_k – vremena završetka operacije i T_p – vremena početka operacije; izraženo u sekundama). Nedostatak ovog pristupa je taj što slanje iole složenih struktura zahteva složena izračunavanja [Roubtsov03]. Drugi problem nastupa ako se podaci šalju kroz više Java *stream*-ova, od kojih svaki može da doda svoje podatke (eng. *overhead*) pri serijalizaciji objekata. U tom slučaju, potrebno je unapred znati količinu podataka koju stream dodaje, pa da se i ti podaci uzmu u obzir pri računanju.

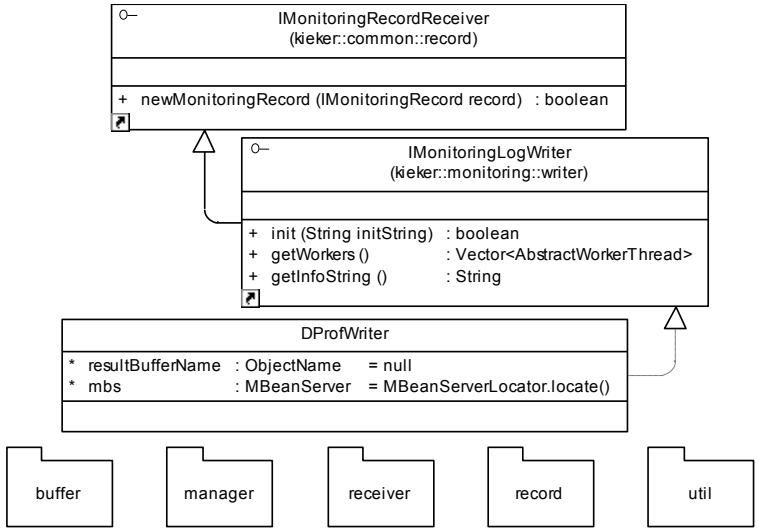
Dostupni Java bazirani alati koji vrše merenje mrežnog protoka se distribuiraju u verzijama za određeni operativni sistem, što pokazuje da se uglavnom koristi prvi pristup.

U slučaju ovog sistema, ostavljeno je korisniku sistema da definiše svoj način merenja u okviru aspekata za praćenje rada aplikacije, a u skladu sa platformom na kojoj se sve izvršava.

3.8 Implementacija komponenti DProf proširenja Kieker okruženja

3.8.1 Beleženje zapisa u Kieker okruženju – `DProfWriter`

Nova komponenta za beleženje zapisa o izvršavanju je *DProfWriter*. Klasa koja predstavlja komponentu, `DProfWriter`, je data na slici 3.16.



Slika 3.16 Dijagram klasa paketa `kieker.monitoring.writer.jmx`

Sa slike se vidi da klasa `DProfWriter` nasleđuje klasu `IMonitoringLogWriter` (a koja implementira `IMonitoringRecordReceiver` interfejs – i klasa i interfejs su standardni deo Kieker distribucije i sve *Monitoring Writer* komponente nasleđuju ovu klasu). Dodata su dva atributa: `mbs` – `MBeanServer` u kom je bafer u koji se smeštaju zapisi i `resultBufferName` – objektno ime pod kojim je bafer registrovan u `MBeanServer`-u.

Da bi se `DProfWriter` klasa koristila u Kieker okruženju, u konfiguracionoj datoteci okruženja se navodi puno ime klase u stavci `monitoringDataWriter`. Deo listinga konfiguracione datoteke okruženja u kojoj se definiše upotreba `DProfWriter`-a dat je na listingu 3.6.

```
1 debug=false
2 monitoringEnabled=true
3 monitoringDataWriter=
    kieker.monitoring.writer.jmx.DProfWriter
```

Listing 3.6 Deo konfiguracione datoteke Kieker okruženja (datoteka *kieker.monitoring.properties*)

Na listingu se vidi da je nakon postavljanja okruženja da skraćeno ispisuje svoje poruke na konzoli (`debug=false`) i uključivanja praćenja (`monitoringEnabled=true`), definisana upotreba `DProfWriter`-a

u procesu praćenja (`monitoringDataWriter=kieker.monitoring.writer.jmx.JMXWriter`).

Pod-paketi koji se nalaze u okviru ovog paketa su: `buffer` (komponenta *ResultBuffer* – pog. 3.8.2), `manager` (komponenta *DProfManager* – pog. 3.8.3), `record` (zapis `DProfMonitoringRecord` – pog. 3.8.4), `receiver` (komponenta *RecordReceiver* – pog. 3.8.5) i `util` (pomoćne klase – pog. 3.8.6).

3.8.2 Komponenta *ResultBuffer*

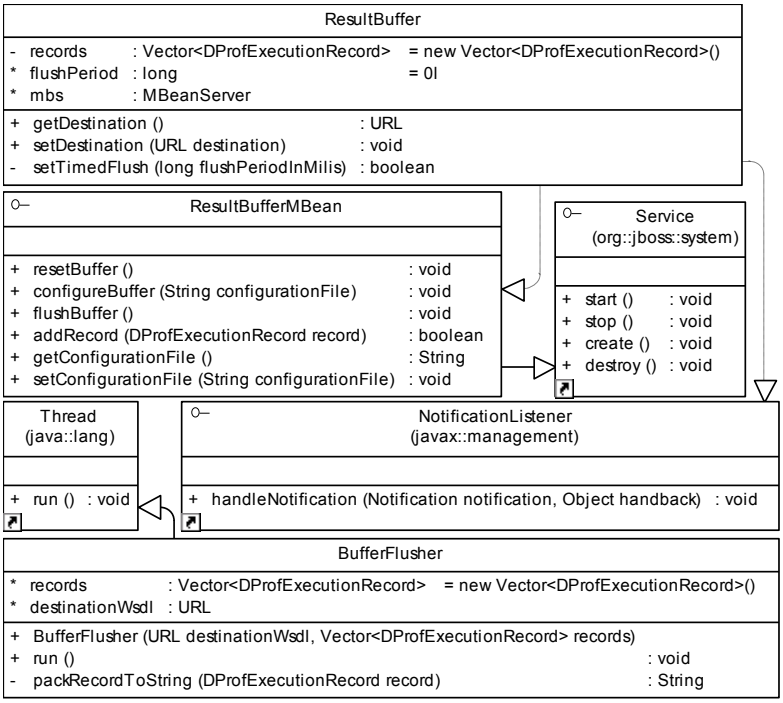
U komponentu *ResultBuffer* komponenta *DProfWriter* komponenta (tj. klasa `DProfWriter`) upisuje *Monitoring Record* zapise (objekte klase `DProfMonitoringRecord` – pog. 3.8.3).

Dijagram klasa paketa `kieker.monitoring.writer.jmx.buffer` u kom se nalaze klase koje čine *ResultBuffer* komponentu dat je na slici 3.17.

Komponenta *ResultBuffer* je realizovana kao *MBean*. Bafer se pokreće kao servis u okviru aplikativnog servera, tj. njegovog kernela.

Upravljački interfejs za ovu komponentu je `ResultBufferMBean`, implementirajuća klasa je `ResultBuffer`.

Interfejs `ResultBufferMBean` ima sledeće metode: `flushBuffer` (metoda koja šalje sve zapisa iz bafera na zadatu lokaciju), `addRecord` (metoda koju `DProfWriter` poziva da doda zapis u bafer), `configureBuffer` (metoda u okviru koje se čita konfiguraciona datoteka bafera – pog. 3.8.2.1 – i vrši konfigurisanje bafera), `resetBuffer` (metoda koja briše sve zapise iz bafera). Metode `getConfigurationFile` i `setConfigurationFile` su ubačene da bi se konfiguraciona datoteka zadala pri instanciranju servisa.



Slika 3.17 Dijagram klasa paketa `kieker.monitoring.writer.jmx.buffer`

Da bi se ovaj MBean pokrenuo kao servis u okviru JBoss aplikativnog servera, klasa mora da implementira metode interfejsa `org.jboss.system.Service`. Da bi primao notifikacije, MBean mora da implementira metode `javax.management.NotificationListener` interfejsa. Notifikacije se koriste da bi se iniciralo periodično slanje zapisa iz bafera. Bafer se registruje da prima periodične notifikacije od JMX *Timer* servisa (ako se bafer tako konfiguriše).

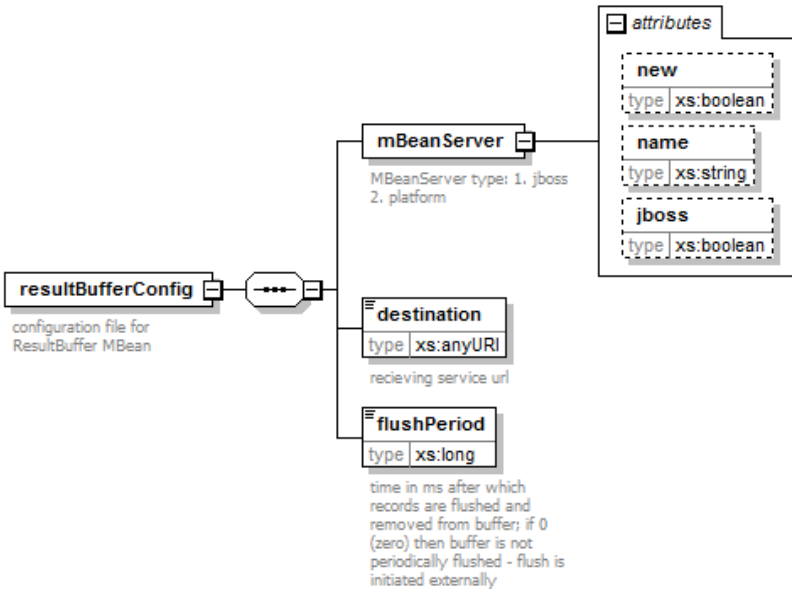
Kada bafer treba da pošalje podatke, oni se proslede objektu klase `BufferFlusher`. Klasa je realizovana kao programska nit (tj. nasleđuje klasu `java.lang.Thread`), da bi `ResultBuffer` nastavio da radi i prima nove zapise, dok `BufferFlusher` šalje stare. Da `ResultBuffer` sam šalje zapise, bio bi blokiran za prijem novih zapisa, dok se svi stari zapisi ne pošalju. Ovako se postupak slanja, koji može da bude vremenski zahtevan, delegira `BufferFlusher`-u. `BufferFlusher` dobija od `ResultBuffer`-a vektor sa zapisima (`Vector<DProfExecutionRecord>`), prepakuje ga u niz stringova i

šalje putem veb-servisa komponenti `RecordReceiver`. Za povezivanje na veb-servis se koriste klase iz paketa `kieker.monitoring.writer.jmx.util.receiverClient` (pog. 3.8.5).

Kada `ResultBuffer` instancira `BufferFlusher`, prosleđuje mu vektor sa zapisima i briše sve zapise iz svoje evidencije.

3.8.2.1 *Konfiguraciona datoteka komponente* ResultBuffer

Komponenta ResultBuffer se konfiguriše pomoću XML konfiguracione datoteke. XML shema za definisanje ove datoteke je data na slici 3.18.



Slika 3.18 XML shema konfiguracione datoteke za komponentu *ResultBuffer*

Prvi korak u konfigurisanju bafera je konfigurisanje registrovanja bafera u `MBeanServer`-u. Bafer može da se registruje u okviru platformskog servera (koji uvek postoji u JVM), može da se kreira novi server ili da se koristi server koji postoji u okviru JBoss aplikativnog servera. Element koji konfiguriše registrovanje je *mBeanServer*. Ovaj element ima tri atributa. Atribut *new* (tipa *boolean*) označava da li se kreira novi server ili se koristi postojeći platformski server. Atribut *name* (tipa *string*) u oba ova slučaja označava ime servera – novog (koji se kreira) ili postojećeg (koji mora da se pronađe u JVM pre registrovanja bafera).

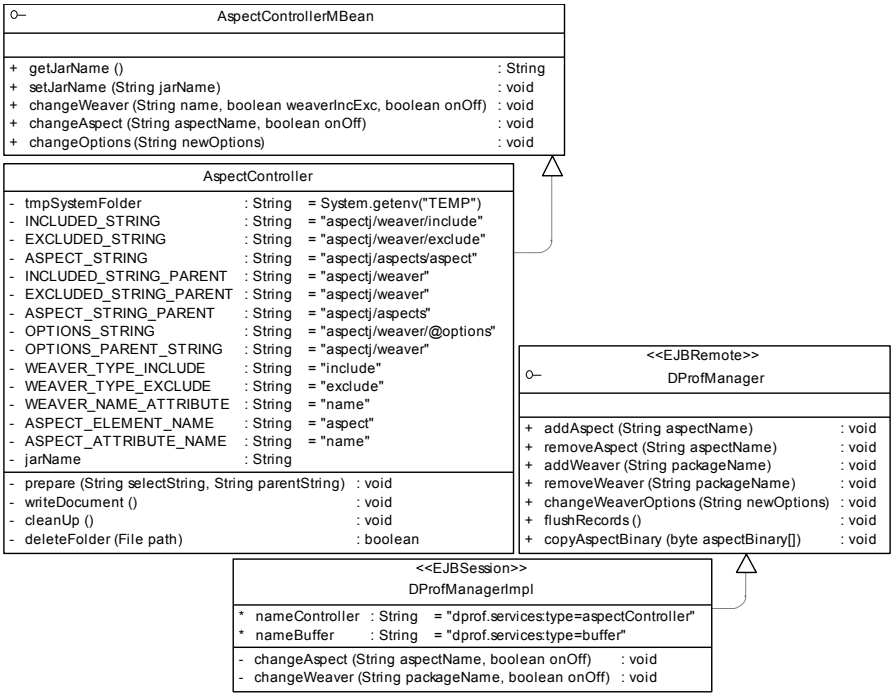
Atribut *jboss* (tipa *boolean*) koji definiše da se bafer registruje u MBeanServer-u JBoss aplikativnog servera. U tom slučaju server se pronalazi pomoću MBeanServerLocator klase (koja je deo JBoss AS distribucije).

Element *destination* označava URL veb-servisa komponente *RecordReceiver*.

Konfigurisanje bafera, da periodično šalje zapise, vrši se preko elementa *flushPeriod*. Zadata vrednost (tipa *long*) predstavlja vreme u milisekundama – period ponavljanja postupka slanja zapisa. Ako je vrednost ovog atributa 0, podaci se šalju samo na zahtev koji stigne preko *DProfManager* komponente.

3.8.3 Komponente *DProfManager* i *AspectController*

U paketu `kieker.monitoring.writer.jmx.manager` se nalaze klase koje implementiraju komponente *ResultBuffer* i *AspectContoller* (slika 3.19).



Slika 3.19 Paket `kieker.monitoring.writer.jmx.manager`

Komponenta *DProfManager* implementirana je kao *stateless session EJB* i anotirana kao veb-servis. Interfejs veb-servisa (i *remote* servis EJB-a)

je `DProfManager`. On sadrži metode za rad sa *AspectController* komponentom – dodavanje i uklanjanje stavki iz *aop.xml* datoteke (aspekata – `addAspect(...)`/`removeAspect(...)`, *weaver* opcija – `addWeaver(...)`/`removeWeaver(...)`) i izmenu *load-time weaving* opcija (`changeWeaverOptions(...)`). Metoda `flushRecords()` se koristi da se *ResultBuffer* komponenti naredi slanje zapisa (ako je bafer konfigurisan da ne šalje periodično, nego po potrebi). Dobavljanje novih aspekt-klasa se vrši pozivom metode `copyAspectBinary(...)`. Implementaciona klasa `DProfManagerImpl` sadrži, pored opisanih metoda iz interfejsa, još dve privatne metode. Sve metode iz interfejsa, koje se odnose na rad sa *AspectController* komponentom, su implementirane tako da pozivaju ove dve metode.

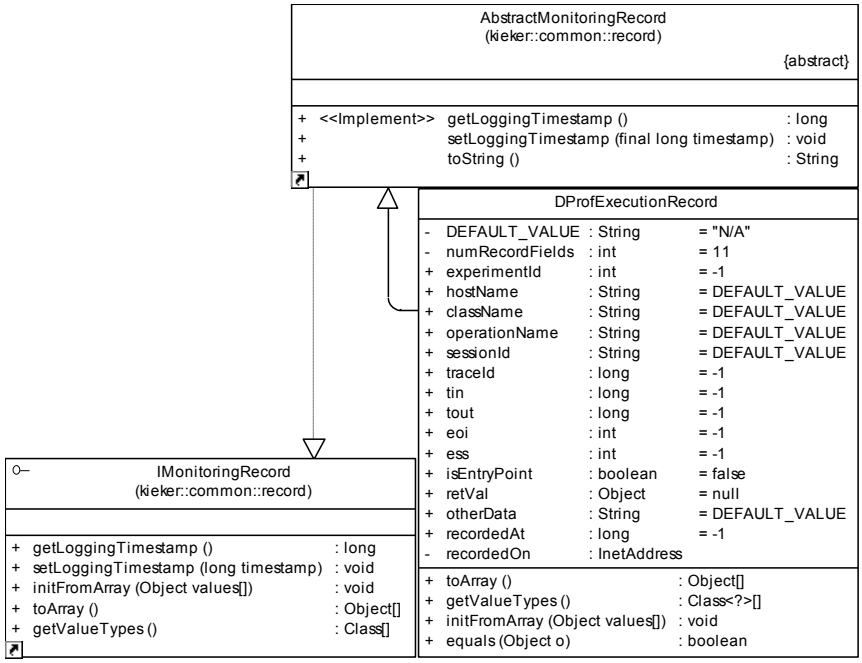
Atributi `nameController` i `nameBuffer` predstavljaju objektna imena pod kojim su komponente *AspectController* i *ResultBuffer* registrovane u *MBeanServer*-u.

AspectController je implementiran kao JMX MBean. Po pristizanju odgovarajućeg zahteva, ova komponenta pristupa *jar* arhivi aplikacije, raspakuje istu i vrši potrebne izmene u *aop.xml* datoteci. Nakon toga, vrši se pakovanje i kopiranje arhive nazad, u odgovarajući folder aplikativnog servera. Arhiva sa kojom *AspectController* radi se zadaje pri pokretanju komponente pomoću `setJar(...)`/`getJar()` metoda. Metoda `changeWeaver(...)` se koristi za izmenu weaver parametara u *aop.xml*, `changeAspect(...)` za izmenu *aspect* parametara, a `changeOptions(...)` za izmenu *load-time weaving* parametara. Privatne metode u implementacionoj klasi (`prepare(...)`, `writeDocument(...)`, `cleanUp(...)` i `deleteFolder(...)`) se koriste da olakšaju rad sa *aop.xml* datotekom.

String konstante definisane u ovoj klasi se koriste za rad sa XPath izrazima koji se koriste u pristupu *aop.xml* datoteci.

3.8.4 Zapis – `DProfMonitoringRecord`

Klasa `DProfMonitoringRecord`, predstavlja zapis koji se koristi u DProf sistemu. Klasa je slična Kieker-ovoj klasi `OperationExecutionRecord`. Dijagram klasa paketa `kieker.monitoring.writer.jmx.record` dat je na slici 3.20.



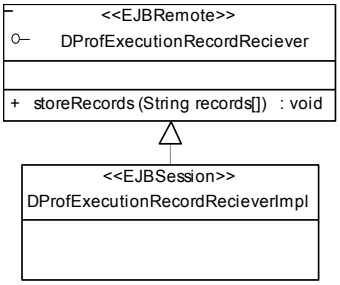
Slika 3.20 Paket `kieker.monitoring.writer.jmx.record`

`DProfExecutionRecord` sadrži sve atribute kao i `OperationExecutionRecord`. Dodatni atributi su `recordedAt` i `otherData`. `recordedAt` označava vreme kada je događaj zabeležen (tačnije, kada je *Monitoring Controller* komponenta kreirala zapis). Na ovaj način dobija se tačno vreme kada se desio što je od značaja za kasniju analizu (npr. povezivanje dobijenih vrednosti sa spoljašnjim događajem za koji znamo da se dogodio u određenom trenutku). Pošto je tip zapisa `OperationExecutionRecord` namenjen samo za merenje dužine izvršavanja metode, atribut `otherData` je dodat da bi se i druge vrste podataka, predviđene za praćenje u DProfSLA dokumentu (pog. 3.9.2), zabeležile u zapis.

3.8.5 *RecordReceiver* komponenta

Prijem zapisa i njihovo smeštanje u relacionu bazu podataka vrši *RecordReceiver* komponenta. Komponenta je realizovana kao *stateless session EJB* i anotirana da bude veb-servis.

Interfejs i implentaciona klasa za ovu komponentu su u paketu `kieker.monitoring.writer.jmx.receiver` prikazanom na slici 3.21.



Slika 3.21 Paket `kieker.monitoring.writer.jmx.receiver`

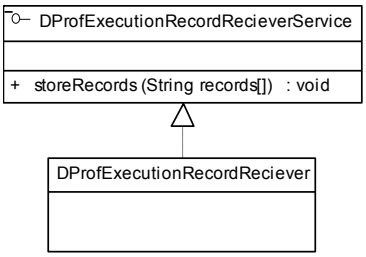
Zapisi se ne šalju u binarnom obliku, nego kao niz stringova. Servis ima samo jednu metodu (`storeRecords(...)`) kojoj se prosleđuje ovaj niz zapisa. Servis prihvata dobijeni niz, parsira dobijene zapise i smešta ih u relacionu bazu.

3.8.6 Pomoćne klase

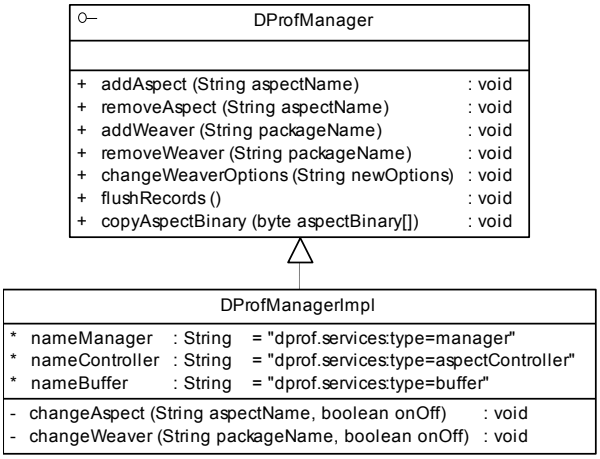
Pomoćne klase su u paketu `kieker.monitoring.writer.jmx.util` na slici 3.22.

Slika 3.22 Sadržaj paketa `kieker.monitoring.writer.jmx.util`

Paketi `kieker.monitoring.writer.jmx.util.receiverClient` (slika 3.23) i `kieker.monitoring.writer.jmx.util.receiverClient` (slika 3.24) sadrže klijentske klase za komunikaciju sa veb-servisima *RecordReceiver* i *DProfManager* komponenata respektivno.

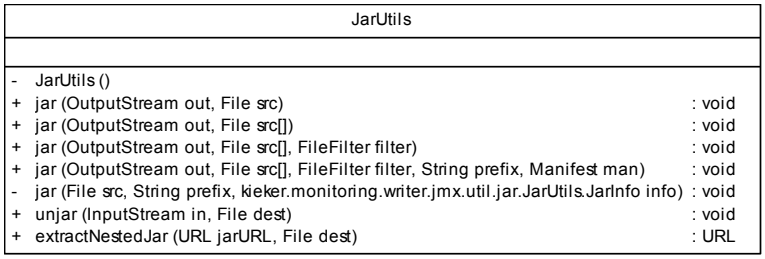


Slika 3.23 Sadržaj paketa `kieker.monitoring.writer.jmx.util.receiverClient`



Slika 3.24 Sadržaj paketa `kieker.monitoring.writer.jmx.util.managerClient`

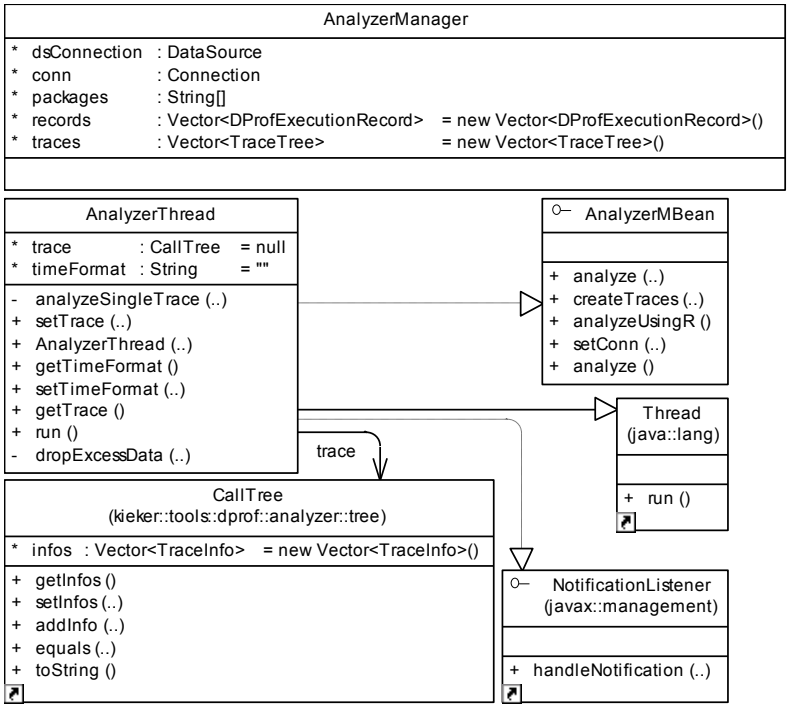
Paket `kieker.monitoring.writer.jmx.util.jar` (slika 3.25) sadrži klasu koja se koristi za rad sa *jar* datotekama. Ova klasa se poziva iz `AspectController` klase, da bi se raspakovala *jar* datoteka sa aplikacijom i ponovo zapakovala, nakon izmene sadržaja *aop.xml* datoteke. `unjar(...)` metoda se koristi za raspakivanje *jar* datoteke, a `jar(...)` metode se koriste za pakovanje.



Slika 3.25 Sadržaj paketa `kieker.monitoring.writer.jmx.util.jar`

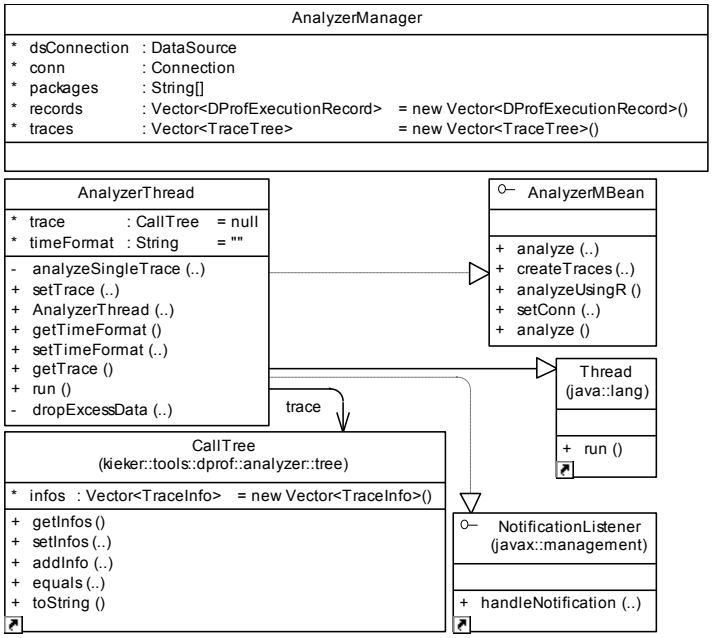
3.9 Analiza zapisa i izmena parametara procesa praćenja

Komponenta *Analyzer* vrši analizu dobijenih zapisa i zadaje novu konfiguraciju procesa praćenja. Klase koje vrše analizu nalaze se u paketu `kieker.tools.dprof.analyzer` (slika 3.26).



Slika 3.26 Sadržaj paketa `kieker.tools.dprof.analyzer`

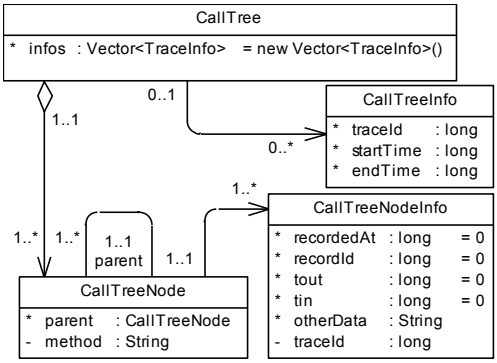
`AnalyzerManager` je implementiran kao servis (implementira interfejs `Service` iz JBoss distribucije). Svaki `AnalyzerManager` je zadužen za jednu aplikaciju (putem odgovarajuće SLA koji mu se prosleđuje). U procesu analize `AnalyzerManager` pokreće programske niti (klasa `AnalyzerThread`) koje vrše analizu pojedinačnih stabala poziva (klasa `TraceTree` iz paketa `kieker.tools.dprof.analyzer.tree`). Koriste se pojedinačne niti da bi se ceo proces paralelizovao i ubrzao, kao što je objašnjeno u poglavlju 3.5.



Slika 3.27 Dijagram klasa u paketu `kieker.tools.dprof.analyzer`

3.9.1 Stablo poziva

Dijagram klasa koje se koriste za reprezentaciju stabla poziva dat je na slici 3.28.



Slika 3.28 Sadržaj paketa `kieker.tools.analyzer.tree`

Klase `Tree` i `TreeNode` su generičke klase. Označavaju stablo i čvor u stablu, respektivno.

Klasa `Tree` ima atribut `rootElement` tipa `TreeNode` koji predstavlja korenski element stabla. Metoda `toString()` daje `String` reprezentaciju stabla. Metoda `equals(..)` se koristi za poređenje dva stabla. Algoritam prolazi rekurzivno kroz celo stablo (pomoću privatne metode `walk(..)`) i poredi čvorove.

`TreeNode` predstavlja čvor stabla. Atribut `data` sadrži podatke čvora, a `children` podređene čvorove. Dodavanje podređenog čvora (na određeno mesto u listi podređenih) se vrši pomoću metode `insertChildAt(..)`, a za uklanjanje `removeChildAt(..)`. Metoda `hasChild()` se koristi za proveru da li je već ubačen isti takav čvor potomak.

Podaci o pozivima aplikacije se čuvaju u stablu poziva, koje je predstavljeno klasom `TraceTree`. Svaki čvor u stablu predstavlja jedan poziv metode i predstavljen je klasom `TraceNode`. Celokupno stablo predstavlja belešku o tome šta se sve izvršava na jedan klijentski zahtev. Čvor u stablu poziva sadrži belešku o tome kada je počeo i kada se završio poziv. Više klijentskih poziva rezultuje istim stablom, tako da je potrebno da uz svako stablo postoji struktura (`java.util.Vector` sa objektima klase `TraceInfo`) koja sadrži informacije o tim pojedinačnim izvršavanjima.

Klasa `TraceTree` nasleđuje generičku klasu `Tree` (tj. konkretizaciju `Tree<String>`) i ima dodatni atribut `infos` – vektor (`java.util.Vector<TraceInfo>`) koji sadrži informacije o pojedinačnim izvršavanjima stabla – `TraceInfo`. `TraceInfo` je praktično beleška o jednom klijentskom pozivu – jednom izvršavanju stabla – i sadrži početno vreme izvršavanja (odgovara početku izvršavanja prvog čvora – predstavljeno atributom `startTime`), krajnje vreme izvršavanja (odgovara završetku izvršavanja poslednjeg čvora u stablu – predstavljeno atributom `endTime`) i identifikacioni broj klijentskog poziva (`traceId`).

Klasa `TraceNode` nasleđuje klasu `TreeNode` (tj. konkretizaciju `TreeNode<String>`). Jedan čvor odgovara jednom zapisu o izvršavanju (`DProfExecutionRecord`-u). Atributi ove klase su vreme početka i završetka izvršavanja (`tin` i `tout`, repsektivno), identifikacija zapisa (`recordId`) i vreme kad je zapis zabeležen (`recordedAt`). Radi lakšeg kretanja kroz stablo dodat je atribut `parent` – čvor predak datog čvora.

3.9.2 Statistička analiza

Programske niti analizatora dobijene podatke za stablo poziva treba da uporede sa SLA. Pošto u dobijenim vrednostima može da bude dosta grešaka merenja, one moraju da se uklone. Ovo se radi pomoću jezika R i

dodatnog paketa *extremevalues*. Tek nakon toga, dobijene vrednosti mogu da se porede sa vrednostima u SLA.

3.9.2.1 Uklanjanje anomalija iz skupa izmerenih vrednosti

Iz skupa izmerenih vrednosti (u odgovarajućim merama) koje su dobijene za generisana stabla poziva, anomalije se uklanjaju pomoću R skripta koji se pokreće u okviru analizatorske niti, datog u listingu 3.7.

```
1 library(extremevalues)
2 podaci = read.table(file="data858132903.txt",
      header=FALSE,skip=1)[,1]
3 outliers = getOutliers(podaci, method="I")$iRight
4 zaCiscenje=podaci
5 zaCiscenje[outliers]=NA
6 cistiPodaci = zaCiscenje[!is.na(zaCiscenje)]
7 write(cistiPodaci,"ociscenRez858132903.txt",
      ncolumns=1)
```

Listing 3.7 R skript za detekciju i uklanjanje anomalija iz skupa izmerenih vrednosti

Svaka analizatorska nit snima podatke u datoteku. Nakon učitavanja extremevalues paketa (linija 1), čitaju se podaci iz dobijene datoteke (linija 2). U datoteci je zapisana string-reprezentacija stabla poziva, a zatim vrednosti. Zato se preskače prvi red pri učitavanju podataka (direktiva `skip`). Funkcija `getOutliers(...)` (linija 3) za zadate podatke i izabrani metod, kao rezultat vraća niz rednih brojeva vrednosti koje su anomalije u zadatom skupu. U linijama 4, 5 i 6 iz učitanih podataka se izbacuju podaci koji su na zadatim dobijenim brojevima. U liniji 7 novi ("očišćen") skup podataka se snima u datoteku. U oba slučaja, datoteke u nazivu sadrže identifikaciju programske niti analizatora.

3.9.3 Kreiranje novih parametara praćenja i nastavak procesa praćenja

Kao što je već opisano, kada se izvrši prvi ciklus praćenja, kreiraju se programske niti i proslede im se stabla poziva. Ako rezultati statističke analize pokažu odstupanje od SLA, novi parametri treba da se dodaju u procesu praćenja. Pošto je cilj analize da se pronađe gde u stablu dolazi do najvećeg odstupanja, prvo se isključuje korenski čvor u stablu. To praktično znači da se *DProfManager* komponenti šalje komanda da u *aop.xml* datoteku doda *exclude* direktivu koja sadrži putanju do prvog čvora.

Ako korenski čvor ima jednog čvora naslednika, dalju analizu dobijenih rezultata sprovodi ista analizatorska programska nit.

Ako korenski čvor ima više čvorova naslednika, za sve njih se kreiraju analizatorske niti, a korenska završava svoj rad.

U novom ciklusu analize, u obzir se uzimaju samo podaci koji su pristigli nakon prethodne analize.

3.10 Priprema okruženja za rad

Celo opisano okruženje se pakuje u dve arhive: arhiva koja se postavlja zajedno sa aplikacijom i koja sadrži komponente za slanje zapisa – *DProfProv.sar* i arhivu koja sadrži komponente koje prihvataju zapise, smeštaju ih u bazu i vrše analizu – *DProfRec.sar*. Obe arhive imaju ekstenziju sar, što je standardna ekstenzija za arhive koje JBoss server pokreće kao servise.

U *DProfProv.sar* se smeštaju komponente *DProfManager*, *AspectController*, *DProfWriter* i *ResultBuffer*. U META-INF folder se dodaje i *jboss-service.xml* datoteka, na osnovu koje će JBoss pokrenuti *AspectController* i *ResultBuffer* MBean komponente. U datoteci se zadaje putanja do konfiguracione datoteke za *ResultBuffer* i putanja do datoteke aplikacije koja se prati (potrebna je *AspectController* komponenti). JMX *Timer* servis koji je potreban *ResultBuffer* komponenti, ovde se definiše, a njegovo konfigurisanje vrši se pri pokretanju same *ResultBuffer* MBean komponente.

Primer konfiguracione datoteke dat je na listingu.

```
1 <server>
2   <mbean code="javax.management.timer.Timer"
       name="dprof.services:type=timer" />
3   <mbean code="kieker.monitoring.writer.jmx.
       buffer.ResultBuffer"
       name="dprof.services:type=buffer">
4     <attribute name="ConfigurationFile">
          ./../server/all/lib/resultBuffer.xml
5     </attribute>
6     <depends>dprof.services:type=timer</depends>
7   </mbean>
8   <mbean code= "kieker.monitoring.writer.jmx.
       manager.AspectController"
       name="dprof.services:type=aspectController
       ">
9     <attribute name="JarName">
```

```
            ./../server/all/deploy/WS_SCM.jar
        </attribute>
10          <depends>dprof.services:type=buffer</d
            epends>
11    </mbean>
12</server>
```

Listing 3.8 Primer *jboss-service.xml* datoteke za *DProfProv.sar* datoteku

Pored *jboss-service.xml* konfiguracione datoteke, okruženju je potrebna biblioteka sa Kieker okruženjem (stavlja se u *lib* folder servera da bi bilo automatski ubačeno u *classpath*).

U *DProfRec.sar* arhivu se smeštaju *RecordReceiver* i *Analyzer* komponente. Da bi obe komponente mogle da pristupe bazi podataka, u serveru mora da bude pokrenut odgovarajući JDBC servis [JBossDS].

Primer konfiguracione datoteke za *DProfRec.sar* arhivu dat je na listingu.

```
1 <server>
2   <mbean code="javax.management.timer.Timer"
        name="dprof.services:type=timer" />
3   <mbean code="kieker.tools.dprof.Analyzer"
        name="dprof.services:type=analyzer">
4     <attribute name="ConfigurationFile">
          ./../server/all/lib/analyzerSLA.xml
      </attribute>
5     <attribute name="ApplicationName">
          SomeApplication</attribute>
6     <depends>dprof.services:type=timer</depends>
7   </mbean>
8 </server>
```

Listing 3.9 Primer *jboss-service.xml* datoteke za *DProfRec.sar* datoteku

Analyzer komponenti se prosleđuje konfiguraciona datoteka koja sadrži SLA dokument i naziv aplikacije koja treba da se prati. Komponenti je potreban i JMX *Timer* servis, koji se konfiguriše pri pokretanju *Analyzer*-a.

3.11 Baza podataka

Baza podataka koja se koristi u ovom sistemu ima jednu tabelu koja sadrži zapise i jednu koja se koristi u analizi podataka.

U tabeli MONITORINGDATA se čuvaju zapisi. Tabela je slična tabeli koju koriste `SyncDBWriter` i `AsyncDBWriter`, koji su standardni deo Kieker distribucije, ali je proširena sa dve kolone. Originalna tabela je imala kolone koje su odgovarale atributima u klasi `OperationExecutionRecord`, dok nova tabela odgovara novom zapisu – `DProfMonitoringRecord`.

Kolona RECORDEDON predstavlja IP adresu na kojoj je generisan zapis i odgovara istoimenom atributu klase `DProfMonitoringRecord`. Kolona RECORDEDAT predstavlja trenutak kada je zapis nastao i odgovara istoimenom atributu klase `DProfMonitoringRecord`. Sa kolonom OTHERDATA je slično, tj. u nju se smeštaju dodatni podaci, predviđeni konfiguracijom praćenja.

SQL skript za kreiranje ove tabele dat je na listingu 3.10.

```
1 create table MONITORINGDATA (
2    AUTOID             bigint not null,
3    EXPERIMENTID smallint not null,
4    OPERATION          varchar(160) not null,
5    SESSIONID          varchar(34) not null,
6    TRACEID            varchar(34) not null,
7    TIN                bigint not null,
8    TOUT               bigint not null,
9    VMNAME             varchar(40) not null,
10   EXECUTIONORDERINDEX     int not null,
11   EXECUTIONSTACKSIZE int not null,
12   RECORDEDAT         bigint not null,
13   RECORDEDON         varchar(15) not null,
14   OTHERDATA          varchar(1024),
15   primary key (AUTOID)
16 );
```

Listing 3.10 SQL skript za kreiranje tabele MONITORINGDATA

Tabela ANALYZERDATA sadrži evidenciju trenutaka kada je izvršena analiza za određenu aplikaciju. SQL skript za kreiranje ove tabele dat je na listingu 3.11.

```
1 create table ANALYZERDATA (
2    AUTOID             bigint not null,
3    LASTDATE           bigint not null,
4    APPLICATIONNAME    varchar(30) not null,
5    primary key (AUTOID)
```

```
6 );
```

Listing 3.11 SQL skript za kreiranje tabele ANALYZERDATA

U koloni LASTDATE je vreme poslednje analize (u milisekundama), a u koloni APPLICATIONNAME je naziv aplikacije koja se prati i koji odgovara nazivu koji se zadaje u konfiguracionoj datoteci za *Analyzer* komponentu.

3.12 Upotreba DProf sistema na drugim platformama

U ovom radu, prikazano je profajliranje distribuiranih aplikacija pisanih za Java platformu. Upotreba ovog sistema moguća je i za praćenje aplikacija napisanih za neku drugu platformu.

Prema dijagramu na slici slici 3.1 sistem je podeljen na dva dela: deo za prikupljanje podataka i deo za analizu i definisanje novih parametara praćenja. Ova dva dela sistema su povezani pomoću veb servisa, pa sva komunikacija ide kroz XML. Praktično, da bi se ovaj sistem koristio za praćenje rada aplikacija napisanih u drugim programskim jezicima, potrebno je da se podsistem za praćenje implementira u odgovarajućem jeziku i da se on poveže na postojeće komponente za prijem i analizu zapisa.

Implementacija Kieker okruženja i DProf proširenja u drugim programskim jezicima može da izgleda komplikovano na prvi pogled. Ipak, posao se svodi samo na prevođenje iz jednog programskog jezika u drugi, jer Kieker i DProf nisu implementirani uz oslonac na native funkcije ili manipulaciju *byte*-kodom, a i ne vrše uticaj na rad JVM (npr. na učitavanje klasa), već su napisani čisto u Java programskom jeziku.

U zavisnosti od izbora programskog jezika, treba izabrati najbolji način za instrumentaciju. Pošto za većinu modernih programskih jezika postoje AOP implementacije (npr. LOOM.NET za C# [Loom]), pristup koji je opisan u ovom radu može da se primeni i na njih.

4 Prikaz upotrebe DProf sistema

Testiranje sistema sprovedeno je nad nekoliko scenarija. Prvo je sprovedeno praćenje brzine odziva aplikacije na pokrenute na više servera u cilju merenja normalnih vrednosti. Drugi scenario obuhvata praćenje rada aplikacije i traženje odstupanja od definisanog SLA.

4.1 Osnovne postavke scenarija

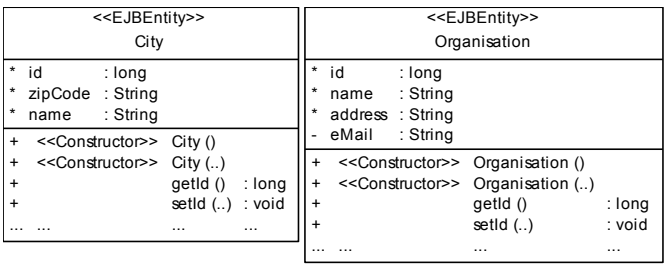
Svi scenariji su zasnovani na praćenju rada aplikacije koja je distribuirana na nekoliko JBoss servera. Serverima se ne pristupa direktno nego preko *load-balancer*-a. Da bi se dobili prihvatljivi rezultati, potrebno je da se prikupi velika količina podataka, dovoljno reprezentativna za statističku analizu rezultata. Velika količina podataka biće generisana velikim broj poziva upućenih klijentskoj strani.

4.1.1 Test aplikacija

Za testiranje sistema iskorištena je SCM aplikacija opisana u [Okanovic06] i [Okanovic08]. Aplikacija se koristi za evidenciju verzija aplikacija (sa svim potrebnim datotekama) i praćenje gde je koja verzija aplikacije instalirana (sa kojim parametrima i dodatnim izmenama).

Reč je o EJB 3.0 [EJB3] baziranoj aplikaciji koja se izvršava na JBoss 5.0 aplikativnom serveru [JBossAS]. Na serverskoj strani, kao sloj za komunikaciju sa bazom podataka su entity EJB. Sloj za pristup *entity* EJB je realizovan pomoću *stateless session* EJB (skr. SLSB), koji su kreirani po *façade* projektnom obrascu [Gamma94]. Ovi SLSB-ovi su ujedno implementirani kao veb-servisi [Kalin09]. Komunikacija između serverske strane aplikacije i klijentske strane se vrši kroz ove veb-servise. Klijentska strana je realizovana kao Java Swing aplikacija [Swing].

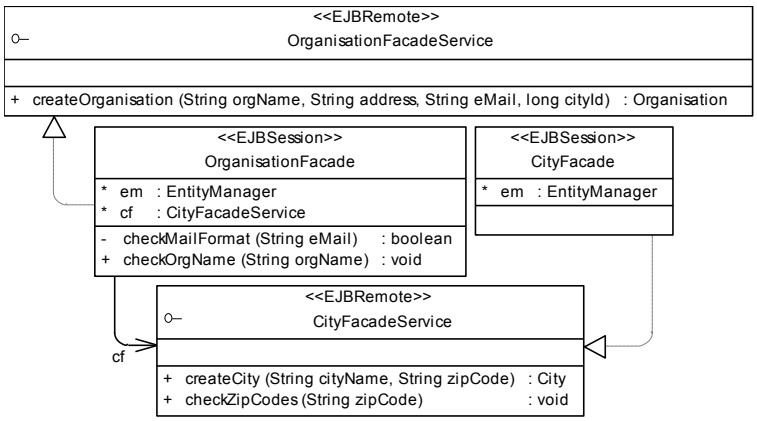
Dijagram klasa *entity* EJB-ova jednog dela distribuirane aplikacije prikazan je na slici.



Slika 4.1 Entity EJB klase u test aplikaciji (jedan deo paketa gint.scm.ws.entity)

Ovde je od interesa samo rad sa organizacijama (u kojima se instaliraju aplikacije – klasa `Organisation`) i gradovima (u kojima se te organizacije nalaze – klasa `City`).

SLSB ("fasade") koji se koriste za pristup ovim *entity* EJB su prikazani na slici.



Slika 4.2 SLSB koji se koriste za pristup *entity* EJB u test aplikaciji (jedan deo paketa `gint.scm.ws`)

Svaki SLSB ima svoj *remote* interfejs i anotiran je kao JAX-WS veb-servis. U okviru `CityFacade` nalaze se metode za kreiranje grada i proveru validnosti poštanskog broja koji je korisnik uneo. U okviru `OrganisationFacade` su metode za kreiranje organizacije, validaciju e-mail adrese i proveru naziva organizacije (da li već postoji u sistemu).

U ovom primeru, klijentska strana postojeće aplikacije nije od interesa, pošto je cilj ovih test-scenarija da se testira distribuirani deo aplikacije – deo koji se izvršava na serveru. Test-scenarija zahtevaju veliki broj poziva upućenih distribuiranom delu aplikacije, pa postojeći *Java Swing* klijent nije pogodan za ovakvo testiranje.

Umesto Java Swing klijenta, napravljena je klijentska Java aplikacija realizovana kao Java klasa u okviru koje se kreiraju programske niti. Svaka programska nit u sebi sadrži petlju (sa zadatim brojem ponavljanja). U svakom ciklusu petlje se poziva slučajno izabrana metoda na serverskoj strani. Na taj način se jednim pokretanjem aplikacije vrši simulacija velikog broja poziva upućenih serverskoj strani.

Kao što je rečeno, aplikacija je postavljena na JBoss aplikativnom serveru. Koristi se *all* konfiguracija, jer se u njoj podržavaju sve funkcionalnosti veb-servisa. Aplikacija je spakovana u jar datoteku

SCM.jar, pa se svim veb-servisima aplikacije pristupa preko WSDL datoteke koja je na adresi: `http://adresa_servera/SCM/naziv_servisa?wsdl`. Npr. za servis za rad sa organizacijama ima WSDL na adresi: `http://somenode.uns.ac.rs/SCM/OrganisationFacade?wsdl` (gde je *somenode.uns.ac.rs* adresa računara na kom je aplikacija postavljena).

4.1.2 Test-platforme

Konfiguracije servera na kojima su sprovedena testiranja su prikazane u tabeli:

Naziv servera	**node1**	**node2**	**node3**	**node4**	**node0**
Procesor:	Intel Core2Duo E8400 (3GHz)				
Memorija:	4GB – podržano 3GB			4GB	6GB
Hard-disk:	160GB				160GB + 1TB
Operativni sistem:	Windows XP Professional (32bit)		Windows 7 Professional (64bit)	Ubuntu 10 (64bit)	Windows 7 Professional (64bit)
IP adresa:	192.168.1.101	192.168.1.102	192.168.1.103	192.168.1.104	192.168.1.100
JBoss server:	5.1.0.GA u *all* konfiguraciji				
Java verzija:	JRE 6.0 za Windows (32bit)			JRE 6.0 za Linux (32bit)	JRE 6.0 za Windows (64bit)
Na serveru se izvršava:	Aplikacija i deo okruženja za generisanje i slanje zapisa				Prihvatanje zapisa i analiza; Apache server sa mod_jk modulom
napomene	operativni sistem podržava samo 3GB memorije				

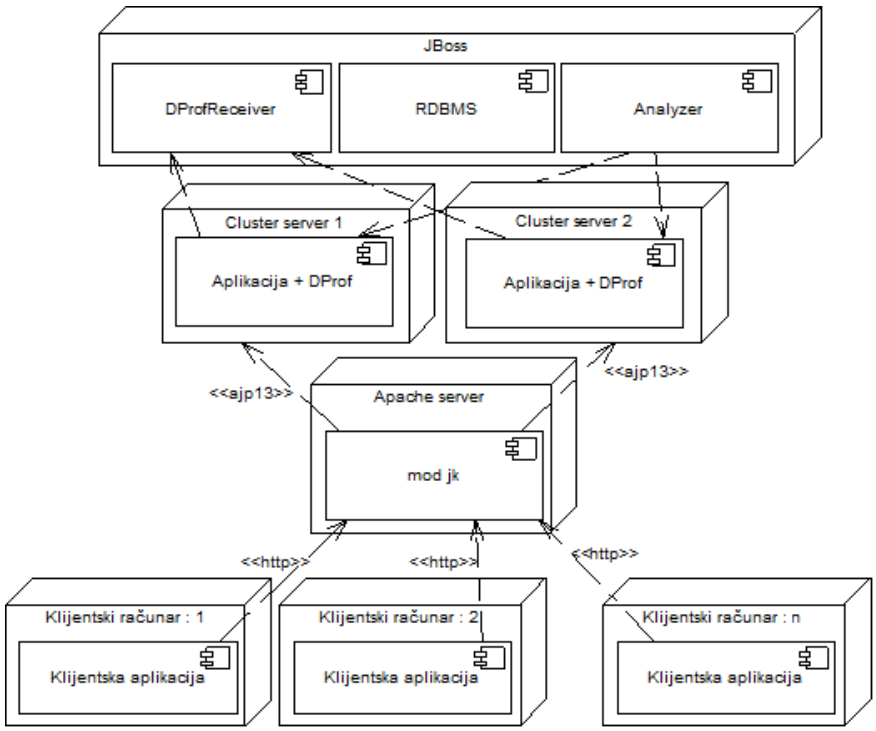
Tabela 4.1 Konfiguracije upotrebljavane u test primerima

4.1.3 Apache *mod_jk*

Aplikacija je konfigurisana da se izvršava na više servera i da se svi pozivi aplikaciji upućuju preko *load-balancer*-a. *Load-balancer* vrši ravnomerno raspoređivanje i usmeravanje poziva, tako da svi serveri budu ravnomerno opterećeni.

Apache *mod_jk* [modjk] je dodatak za Apache server [ApacheServer] pomoću kog se server povezuje na Tomcat servlet kontejner, a koji je

standardni deo JBoss aplikativnog servera. Zbog ovoga, *mod_jk* u okviru Apache servera, može da se koristi za raspoređivanje poziva prema aplikaciji koja se izvršava na više JBoss servera (eng. *cluster*). Prikaz jedne ovakve konfiguracije (sa dva servera na koje se raspoređuju pozivi) prikazan je na slici.



Slika 4.3 Primer konfiguracije sistema za praćenje aplikacije koja se izvršava na grupi servera sa *mod_jk load balancer*-om

Na JBoss serverima koji su u cluster-u nalazi se aplikacija sa okruženjem za praćenje. Klijentski zahtevi stižu na server na kom je Apache sa *mod_jk* proširenjem i *mod_jk* ih usmerava na neki od servera. Putem AJP13 protokola, *mod_jk* upravlja Tomcat kontejnerima u okviru JBoss servera. Okruženja na ovim serverima šalju (putem veb-servisa) dobijene zapise *DProfReceiver* komponenti koja je na posebnom JBoss serveru. Na tom serveru su i baza podataka, u koju se smeštaju zapisi, i *Analyzer* komponenta. *Analyzer* analizira zapise i šalje (putem veb-servisa) nove parametre praćenja okruženjima na serverima koji su u *cluster*-u.

U okviru *mod_jk* konfiguracionih datoteke navode se aplikacije (tj. njihovi konteksti, *root* putanje) čiji pozivi se raspoređuju serverima, tako da treba u njih da se ubaci putanja do test aplikacije, tj. "/SCM/*".

Svaki od ovih čvorova (osim *node0*), treba da se navede u *workers.properties* datoteci *mod_jk* proširenja. Primer datoteke (jedan deo) prikazan je na listingu 4.1.

```
1 worker.list=loadbalancer
2 worker.node1.port=8009
3 worker.node1.host=192.168.1.101
4 worker.node1.type=ajp13
5 worker.node1.lbfactor=1
6 worker.node2.port=8009
7 worker.node2.host=192.168.1.103
8 worker.node2.type=ajp13
9 worker.node2.lbfactor=1
10 worker.loadbalancer.type=lb
11 worker.loadbalancer.balance_workers=node1,node2
```

Listing 4.1 Primer *workers.properties* datoteke sa dva servera

U ovom primeru navedena su dva servera (192.168.1.101 i 192.168.1.103) sa kojima *mod_jk* komunicira preko porta 8009 i protokola AJP13. Ostali serveri se ubacuju u listu na isti način.

4.2 Praćenje rada softvera

Praćenje brzine odziva sprovedeno je pri izvršavanju aplikacije na nekoliko kombinacija servera. Cilj je bio da se utvrdi koliki je dobitak na performansama pri dodavanju novih servera. Prvo je izvršeno merenje sa uključena sva četiri servera, zatim sa tri (*node1*, *node2*, *node3*), sa dva (*node1*, *node2*) i na kraju samo sa uključenim jednim serverom (*node1*). Nije vršena nikakva adaptacija nego su merena vremena odziva u situacijama kada serverima pristupa 10, 20, 30, 50, 75, 100 i 150 klijenata.

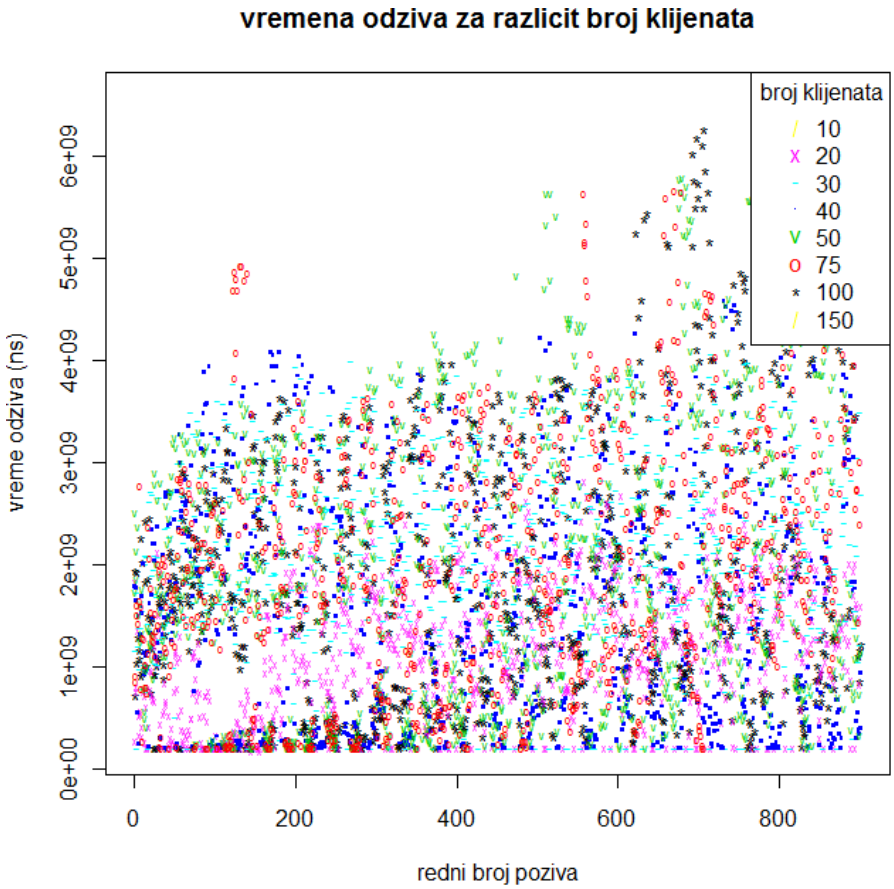
Aspekt koji je upotrebljen je `OperationExecutionAspectAnnotationServlet` koji je deo standardne Kieker distribucije. U okviru ovog aspekta prvo se očita vreme pre izvršavanja (sa `currentTimeNanos()`). Zatim se pozove `proceed()` da bi se izvršila operacija koju je aspekt presreo. Na kraju se očitava vreme završetka i kreira zapis. *aop.xml* datoteka u kojoj se ovo definiše data je u listingu.

```
<aspectj>
 <weaver options="-verbose -showWeaveInfo">
  <include within="gint.scm.ws..*" />
  <include within="gint.scm.ws.entity..*"/>
 </weaver>
 <aspects>
  <aspect name= "kieker.monitoring.probe.
    aspectJ.executions.
    OperationExecutionAspectAnnotationServlet"/>
 </aspects>
</aspectj>
```

Listing 4.2 *aop.xml* datoteka za prvi primer

4.2.1 Analiza dobijenih rezultata

Dobijeni rezultati, za metodu `OrganisationFacade.createOrganisation()` prikazani su na grafiku na slici 4.4. Rezultati za ostale metode su slični.

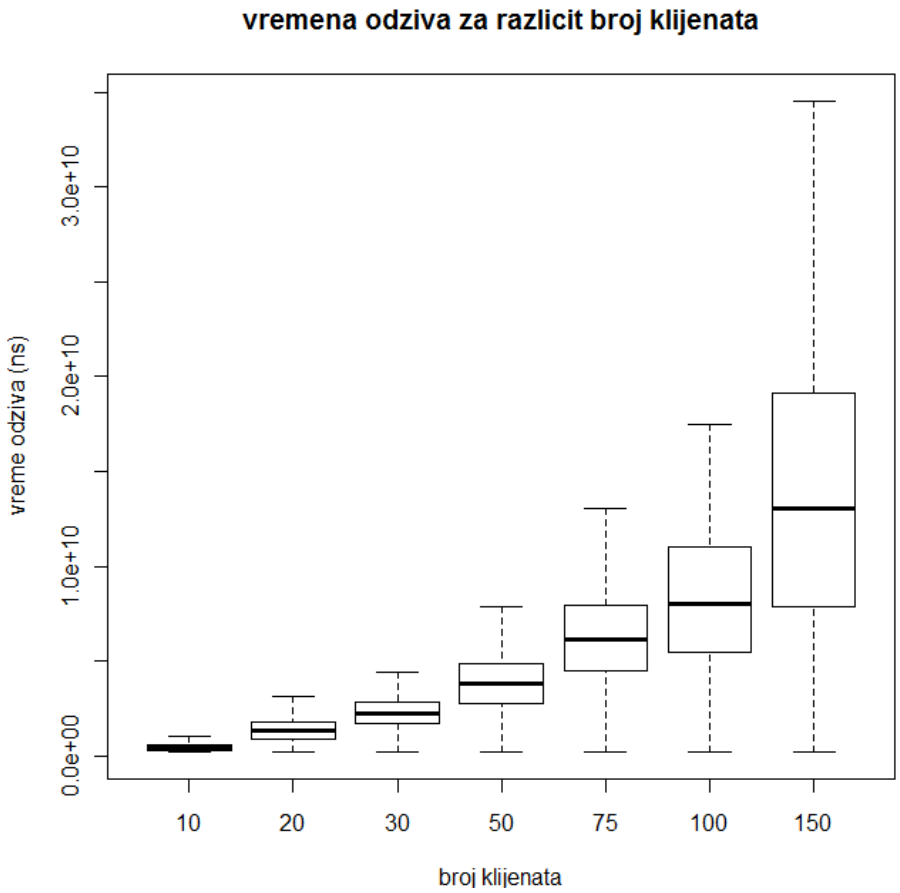


Slika 4.4 Primer rezultata merenja trajanja izvršavanja poziva funkcije `OrganisationFacade.createOrganisation()` na jednom serveru za različit broj klijenata

Slika prikazuje rezultate za konfiguraciju u kojoj radi samo jedan od servera (konkretno, *node4*).

Ovakvi rezultati ne mogu mnogo da nam pokažu zbog velikog broja ekstremnih vrednosti. Ekstremne vrednosti su ovde mnogo veće (za više od jednog reda veličine) od ostalih vrednosti. Zbog njih, npr. dobijene srednje vrednosti ne daju realnu sliku i ne može da se napravi adekvatno poređenje između merenja.

Ako primenimo skript napisan u programskom jeziku R za uklanjanje ekstremnih vrednosti, dobija se nov, pročišćen, skup vrednosti. Novi, pročišćeni rezultati su prikazani na slici 4.5 u *box plot* formatu.

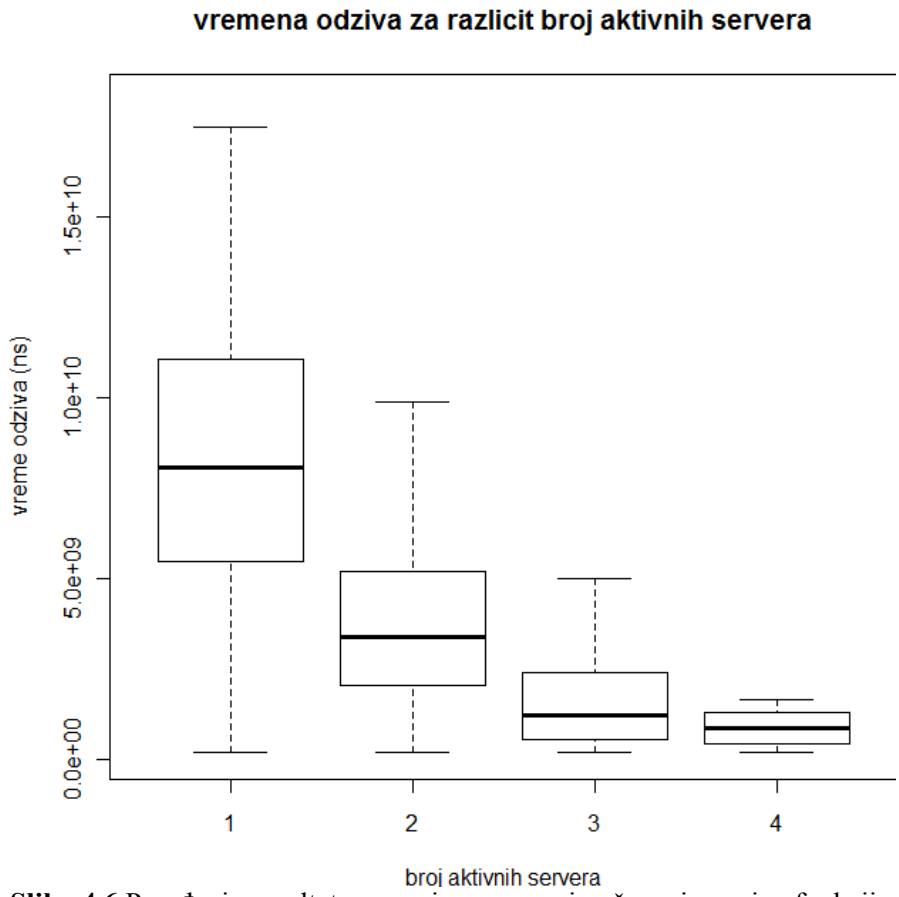


Slika 4.5 Rezultati merenja vremena izvršavanja poziva funkcije `OrganisationFacade.createOrganisation()` na jednom serveru u različitim konfiguracijama, očišćeni od ekstremnih vrednosti

Sa slike se vidi da vreme odziva raste sa brojem klijenata, kao što je očekivano.

Ovakvi, pročišćeni, rezultati daju bolju sliku o tome kako se menja vreme odziva sa brojem klijenata.

Poređenje vremena odziva za različite konfiguracije servera dato je na slici 4.6.



Slika 4.6 Poređenje rezultata merenja vremena izvršavanja poziva funkcije `OrganisationFacade.createOrganisation()` za različite konfiguracije servera u slučaju kada im pristupa 75 klijenata

Sa slike može da se vidi da su dobijeni rezultati o;ekivani i u skladu sa Amdalovim zakonom: dodavanje novog servera u konfiguraciju daje brži odziv aplikacije [Amdahl67].

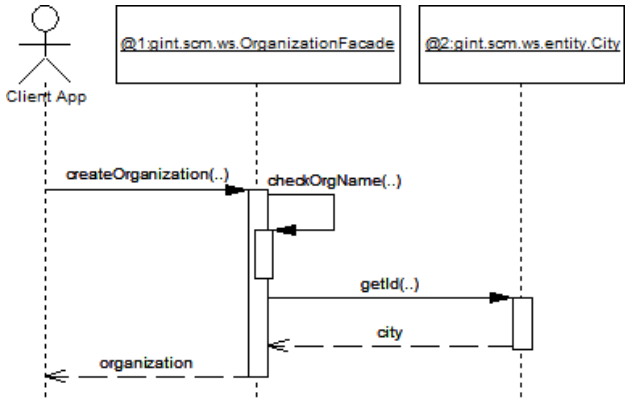
Dobijeni rezultati pokazuju veću stabilnost vremena odziva u slučaju upotrebe više servera, tj. standardna devijacija vremena izvršavanja se takođe smanjuje.

Sve ovo navodi na zaključak da je dodavanje novih servera u sistem dovelo do bržeg odziva i veće garancije stabilnosti rada, što je i očekivani rezultat.

4.3 Utvrđivanje poštovanja SLA

Konfigurisanje sistema za praćenje poštovanja SLA biće prikazano ponovo na praćenju zauzeća memorije.

U okviru test aplikacije, posmatra se deo aplikacije prikazan dijagramom sekvenci sa slike 4.7.



Slika 4.7 Stablo poziva koje se posmatra u test-primeru

Prethodno su utvrđene prosečne vrednosti za zauzeće memorije tokom izvršavanja ovog dela aplikacije i uvrštene su u SLA koji je dat u listingu 4.3.

```
1 <DProfSLA>
2  <Parties><Provider name="...">...</Provider>
3  <Consumer name="...">...</Consumer></Parties>
4    <Trace metric="usedMemoryPct"
        name="[{gint.scm.ws.OrganisationFacade.
        createOrganization,[{gint.scm.ws.
        OrganizationFacade.checkOrgName,[]},
        {gint.scm.ws.entity.City.getId,[]}]"
        nominalValue="52">
5    <Trace metric="usedMemoryPct"
        name="[{gint.scm.ws.Organization.
        checkOrgName,[]}]" nominalValue="52.3"/>
6    <Trace metric="usedMemoryPct"
        name="[{gint.scm.ws.entity.City.getId,[]}]"
        nominalValue="52.5"/></Trace>
7  <Timing>
8    <SamplingPeriod>3600000</SamplingPeriod>
9 </Timing></DProfSLA>
```

Listing 4.3 Sadržaj SLA za praćenje zauzetosti memorije u test aplikaciji

SLA je definisan tako da se analiza rezultata vrši na svakih sat vremena.

Deo koda aspekta koji se koristi u procesu praćenja dat je u listingu 4.4.

```
1 @Aspect
2 public class MemoryProfilingAspect extends
     AbstractOperationExecutionAspect {
3   // ...
4   private static Sigar sigar = new Sigar();
5   @Pointcut ("execution (@kieker.monitoring.
       annotation.OperationExecutionMonitoringProbe
       java.lang.Object *(..))")
6   public void monitoredMethod() {}
7   @Around("monitoredMethod()")
8   public Object doBasicProfiling(
       ProceedingJoinPoint thisJoinPoint) throws
       Throwable {
9     // ...
10    double memory = 0;
11    try {
12     Object retVal = proceed();
13     // merenje zauzeca memorije
14     memory = sigar.getMem().getUsedPercent();
15     // ...
16    } catch (Exception e) {
17     throw e; // izuzeci se prosledjuju
        aplikaciji
18    } finally {
19     DProfExecutionRecord dProfExec = new
        DProfExecutionRecord(...);
20     return execData.retVal;
21    }
22  }
23 }
```

Listing 4.4 Aspekt koji se koristi za praćenje zauzetosti memorije u test aplikaciji

Za merenje zauzetosti memorije, u ovom aspektu, koristi se paket SIGAR [SIGAR].

Merenje zahteva inicijalizaciju SIGAR paketa (linija 4). Nakon toga može da se vrši merenje, kao što je prikazano u okviru linije 14. U ovom slučaju pozove se metoda `sigar.getMem()`, koja kao rezultat vraća

objekat klase `Mem` koji sadrži informacije o sistemskoj memoriji. Pozivom `sigar.getMem().getUsedPercent()` očitava se procenat zauzete memorije.

SIGAR se koristi zato što ima mogućnost merenja više parametara nego u slučaju kada se koriste platformske MXBean komponente. Kada bi koristili MXBean komponente u liniji 4 sa listinga 4.4 bi stajao kod sa listinga 4.5.

```
14 memory = ManagementFactory.getMemoryMXBean().
     getMemoryUsage().getUsed();
```

Listing 4.5 Upotreba MemoryMXBean-a za merenje zauzetosti memorije u test aplikaciji

Inicijalni konfiguracija sistema za praćenje nalazi se u *aop.xml* datoteci, čiji sadržaj je dat u listingu 4.6.

```
 1 <aspectj>
 2   <weaver>
 3     <exclude within="kieker..*" />
 4     <exclude within="java..*, javax..*,
           org.jboss..*" />
 5     <include within="org.gint.scm.ws..*">
 6     <exclude
           within="org.gint.scm.ws.OrganisationFacade.
           checkOrgName">
 7     <exclude
           within="org.gint.scm.ws.entity.City.getId"
 8   </weaver>
 9   <aspects>
10     <aspect name="kieker.monitoring.probe.dprof.
           MemoryProfilingAspect"/>
11   </aspects>
12 </aspectj>
```

Listing 4.6 Sadržaj aop.xml datoteke kojom se konfiguriše praćenje test aplikacije

Sistem počinje sa praćenjem uključenim samo na prvom nivou (tj. prati se samo potrošnja memorije tokom izvršavanja metode `OrganisationFacade.createOrganization()`). Tek ako se utvrdi da postoji povećana potrošnja memorije na ovom nivou, uključuje se praćenje i na drugom nivou (metode `Organization.checkOrgName` i `City.getId`), tako što se uklone linije 6 i 7 iz *aop.xml* datoteke.

Dobijeni rezultati su prikazani u tabeli 4.2.

Metoda koja se prati / *Procenat zauzete memorije*	`Organization.createOrganization()`	`City.getId()`	`Organization.checkName()`
Praćenje samo na prvom nivou	52.25557	Nije praćeno	Nije praćeno
Praćenje na dva nivoa	52.25658	52.25657	52.55579

Tabela 4.2 Rezultati praćenja poštovanja SLA

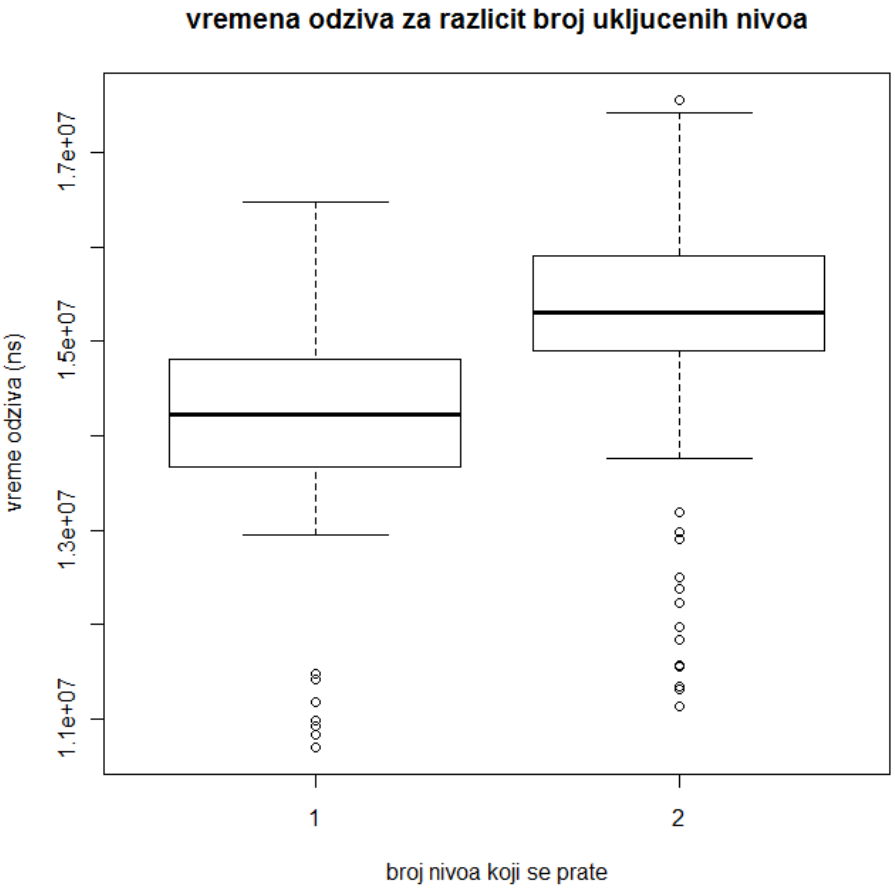
Rezultati u prvih sat vremena praćenja pokazali su da se tokom izvršavanja metode `Organization.createOrganization()` procenat zauzete memorije povećava preko nivoa predviđenog SLA – 52.25557% prema predviđenih 52%.

Nakon što je uključeno praćenje i na drugom nivou, podaci su ponovo analizirani posle sat vremena. Analiza je pokazala da je potrošnja memorije tokom izvršavanja metode `Organization.checkOrgName` veća nego što je predviđeno – 52.55579% prema predviđenih 52.5% – i ta metoda zahteva analizu i, eventualnu preradu.

Ako se gleda izmerena potrošnja memorije tokom izvršavanja metode `Organization.createOrganization()`, vidi se da je došlo do malog povećanja kad je uključeno praćenje na drugom nivou – sa 52.25557% na 52.25658%.

Vreme odziva se takođe povećalo. Dobijeni rezultati su prikazani na slici 4.8.

Da bi smanjio zauzeće resursa, sistem, u normalnom režimu prati samo metode na prvom nivou stabala. Rezultati pokazuju da se na taj način zauzima i manje memorije i da je odziv aplikacije brži. Tek ako se javi neki problem (ako se utvrdi da postoji razilaženje od SLA), uključuje se praćenje na dodatnim nivoima. Takođe, sistem je projektovan tako da se dodatni nivoi uključuju sukcesivno, a ne svi odjednom. Ne traži se uzrok u svim granama odjednom, nego samo u onoj u kojoj postoji odstupanje.



Slika 4.8 Poređenje trajanja izvršavanja poziva ako se prati samo jedan nivo i ako se uključi praćenje i drugog nivoa u stablima poziva

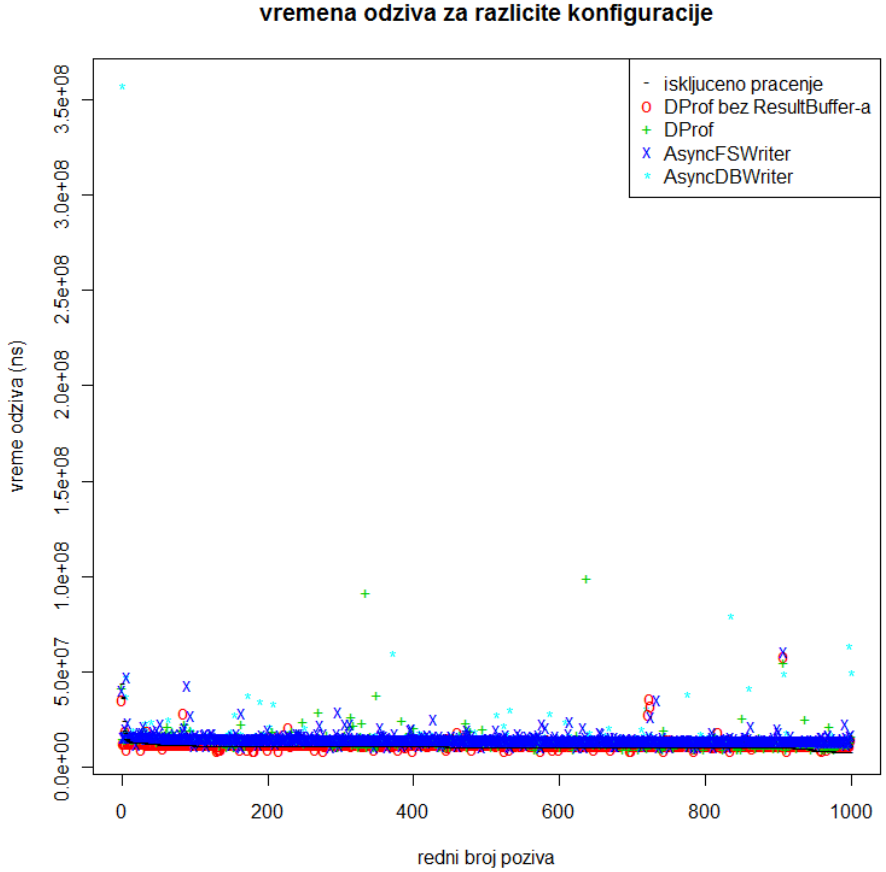
4.4 Poređenje uticaja DProf sistema i drugih Kieker komponenti na softver koji se prati

Da bi analizirali uticaj DProf sistema na softver, kreiran je scenario u kom se vrši hiljadu poziva aplikaciji i meri trajanje izvršavanja tih poziva na klijentskoj strani. Merenja su vršena u sledećih pet situacija:

1. Merenje odziva kada je sistem za praćenje isključen. Na ovaj način se meri standardan odziv aplikacije.
2. praćenje pomoću DProf sistema, ali bez *ResultBuffer*-a – u ovom slučaju *DProfWriter* samo vrši merenje odziva, ali rezultate ne šalje u *ResultBuffer*

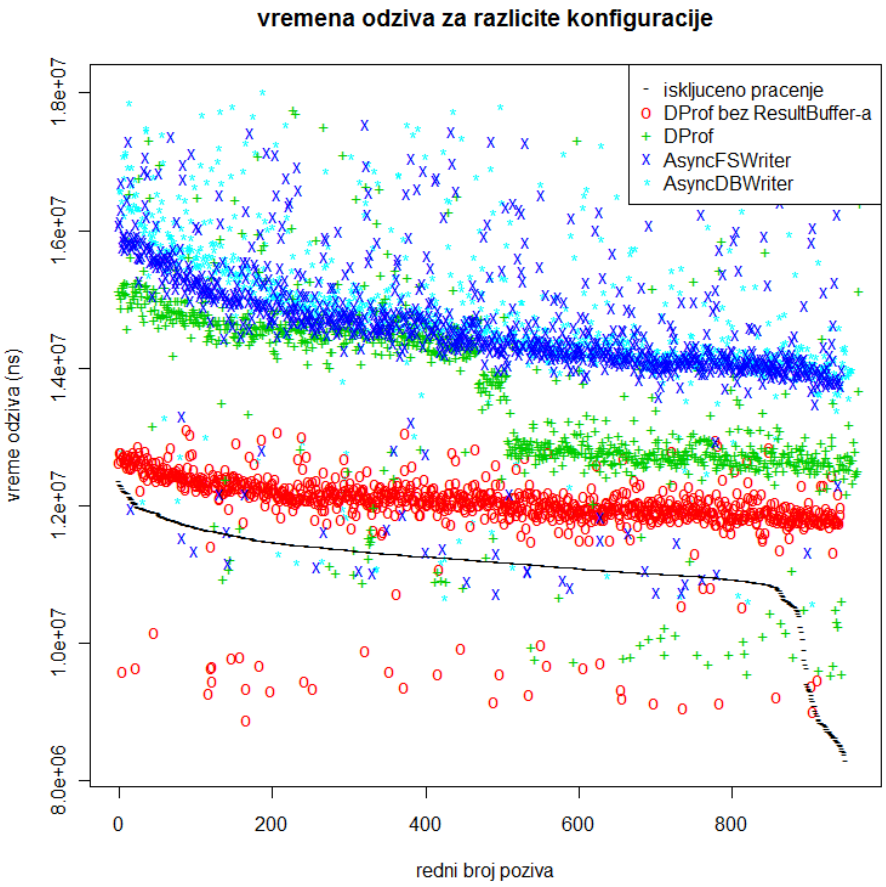
3. praćenje sa DProf sistemom – ista situacija kao u prethodnom slučaju, samo što *DProfWriter* ovde upisuje podatke u *ResultBuffer*
4. uključeno merenje sa Kieker-ovim originalnim `AsyncFSWriter`-om, koji upisuje podatke u datoteku na hard-disku,
5. uključeno merenje sa Kieker-ovim originalnim `AsyncDBWriter`-om, koji upisuje podatke u bazu podataka.

Dobijene vrednosti za svih pet slučajeva su prikazane na slici 4.9.



Slika 4.9 Vremena odziva pri upotrebi različitih Monitoring Writer-a i pri isključenom praćenju

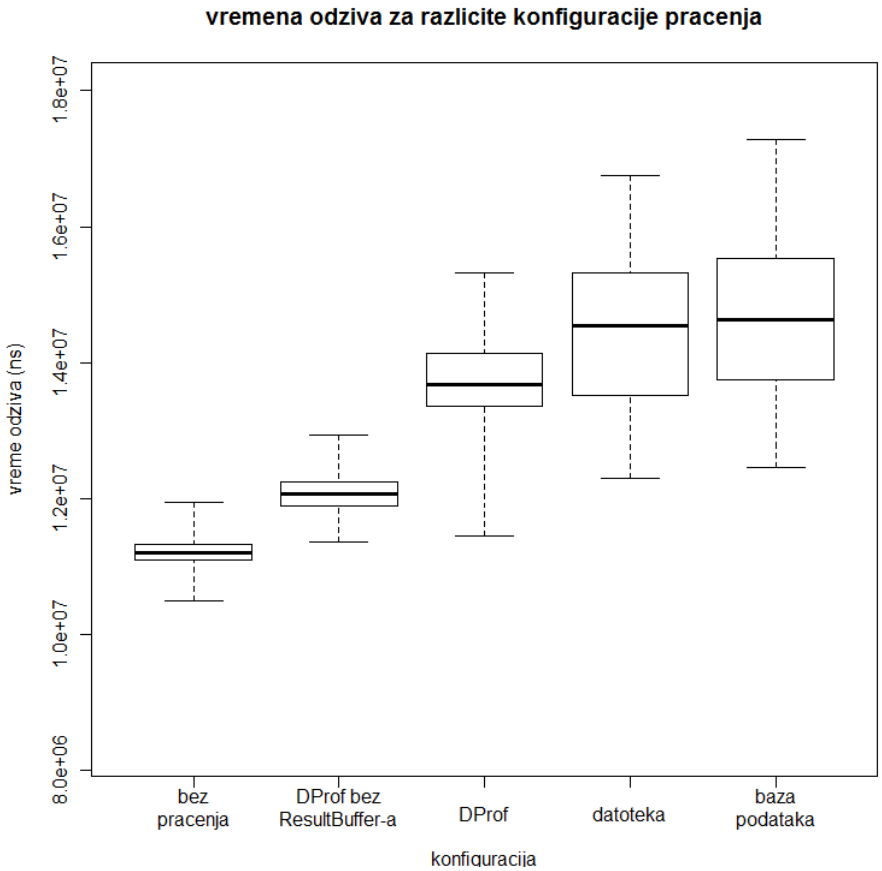
Jasno je da bi ovi podaci mogli dalje da se analiziraju, moraju da se očiste od ekstremnih vrednosti. To je ponovo urađeno pomoću R skripta sa listinga 3.6. Nakon uklanjanja, dobijaju se rezultati prikazani na slici 4.10.



Slika 4.10 Rezultati vremena odziva pri upotrebi različitih Monitoring Writer-a i pri isključenom praćenju, očišćeni od ekstremnih vrednosti

Rezultati pokazuju da vreme odziva opada sa brojem poziva, što je očekivano. Nagli padovi (koji se javljaju u konfiguraciji u kojoj je praćenje isključeno i u kojoj se praćenje vrši pomoću DProf sistema), su posledica rada JVM i aplikativnog servera. Ovi padovi bi se verovatno javili i u ostalim konfiguracijama, ako bi pustili da se dovoljno dugo izvršavaju.

Dalje poređenje vremena izvršavanja izvršeno je na slici 4.11.



Slika 4.11 Poređenje srednjih vremena odziva i standardnih devijacija pri upotrebi različitih Monitoring Writer-a i pri isključenom praćenju

Srednje vreme izvršavanja izabrane metode je najmanje kada je praćenje potpuno isključeno i iznosi 11133404ns. Ako se uključi praćenje pomoću DProf sistema, ali se rezultati ne upisuju u *ResultBuffer*, nego se samo vrši merenje, srednje vreme izvršavanja je 12000103ns. Uključivanje *ResultBuffer*-a povećava srednje vreme izvršavanja na 13600825ns. Vreme odziva pri upotrebi DProf sistema je manje nego pri upotrebi Monitoring Writer-a koji su deo standardne Kieker-ove distribucije – AsyncFSWriter-a i AsyncDBWriter-a. Srednja vremena odziva za ova dva Monitoring Writer-a su 14624948ns i 14790968ns, respektivno.

Standardna devijacija vremena odziva je manja pri upotrebi DProf sistema nego pri upotrebi Kieker-ovih Monitoring Writer-a. To znači da se uz upoterbu DProf sistema dobija stabilniji odziv.

Dobijeni rezultati pokazuju da upotreba DProf sistema za praćenje aplikacije, iako ima uticaj na performanse, on je manji od uticaja koji imaju originalni Kieker-ovi pisači.

5 Predikcija ponašanja na osnovu dobijenih rezultata

Podaci koji se dobijaju u procesu praćenja se obično koriste da ukažu na probleme u sistemu. U prethodnom poglavlju pokazano je kako je moguće pomoću sistema za praćenje aplikacija DProf pronaći koji deo softvera ne funkcioniše prema očekivanjima.

Kao što je navedeno u uvodu, podaci dobijeni praćenjem mogu da se upotrebe za predviđanje rada aplikacije. Primena DProf sistema biće prikazana u dva slučaja. Prvo će biti prikazano kako podaci koji su prikupljeni praćenjem mogu da se iskoriste za predviđanje da li i kada je potrebno da se dodaju u sistem novi resursi. Drugi primer upotrebe biće primena podataka dobijenih pomoću DProf sistema za planiranje podmlađivanja softvera.

5.1 Promena odziva aplikacije pri promeni broja korisnika i servera

U ovom primeru, podaci dobijeni praćenjem pomoću DProf sistema koriste se za procenu zavisnosti vremena odziva aplikacije od broja klijenata koji pristupaju aplikaciji i od broja servera na kojima se aplikacija izvršava.

U postupku procene koriste se podaci dobijeni u poglavlju 4.2 na test platformi prikazanoj u 4.1.2. Srednje vrednosti vremena odziva za različite konfiguracije servera prikazane su u tabeli 5.1.

Tabela 5.1 Srednja vremena odziva za različite konfiguracije praćenja

	node1	*node2*	*node3*	*node4*	*node3 + node4*	*node2 + node3 + node4*	*node1+ node2 + node3 + node4*
10	2.480E+08	2.541E+08	4.776E+08	4.655E+08	2.646E+08	3.108E+08	2.643E+08
20	6.625E+08	7.421E+08	1.208E+09	1.368E+09	7.231E+08	3.686E+08	3.126E+08
30	1.857E+09	1.953E+09	2.113E+09	2.257E+09	1.556E+09	5.644E+08	4.132E+08
50	4.481E+09	4.876E+09	3.527E+09	3.878E+09	2.078E+09	1.262E+09	5.467E+08
75	4.536E+09	4.488E+09	5.984E+09	6.433E+09	3.033E+09	2.478E+09	1.215E+09
100	6.509E+09	6.460E+09	8.396E+09	9.051E+09	3.672E+09	3.339E+09	1.603E+09
150	1.053E+10	1.041E+10	1.338E+10	1.466E+10	6.197E+09	5.743E+09	3.167E+09

Sa slike 5.1, na kojoj je dat grafički prikaz ovih rezultata, se jasno vidi da sa porastom broja klijenata koji pristupaju posmatranoj aplikaciji raste vreme odziva. Takođe, vreme odziva opada sa porastom broja servera na kojima se aplikacija izvršava.

Da bi se izvršila procena kako na vreme odziva utiče broj klijenata i broj servera, upotrebljena je regresiona analiza [Rutherford06]. Za analizu je upotrebljen alat SPSS 15 [Cronk08].

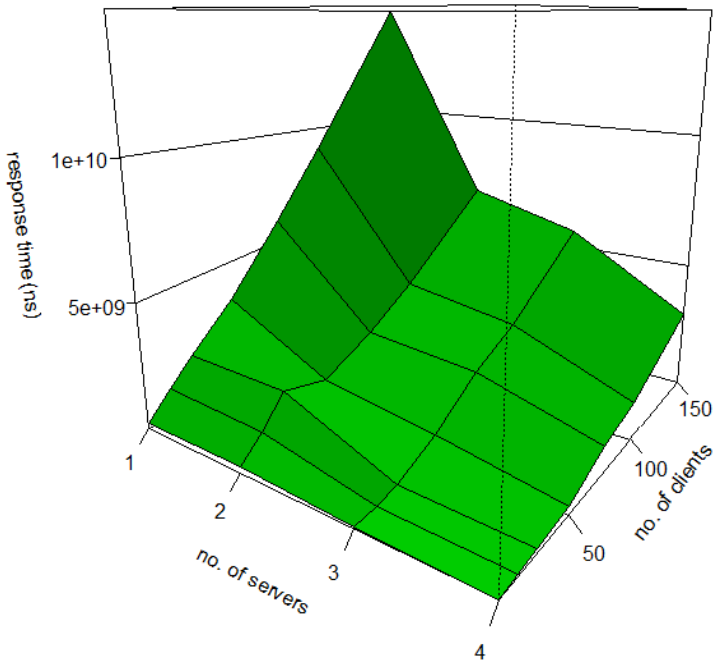
Broj servera i broj klijenata su nezavisne promenljive (prediktorske) u modelu, a srednje vreme odziva je zavisna promenljiva.

Rezime modela je dat u tabeli 5.2.

Tabela 5.2 Rezime modela linearne regresije za predikciju vremena odziva

R	R^2	Adjusted R Square	Standardizovana greška estimacije
0,913	***0,834***	0,826	$1{,}478 \cdot 10^9$

U tabeli se vidi da prediktorske promenljive "objašnjavaju" 83.4% vrednosti zavisne promenljive, što je statistički odličan rezultat.



Slika 5.1 Poređenje srednjih vremena odziva i standardnih devijacija pri upotrebi različitih Monitoring Writer-a i pri isključenom praćenju

U tabeli 5.3 prikazana je analiza varijanse (ANOVA).

Tabela 5.3 Analiza varijanse (ANOVA tabela) za model linearne regresije za predikciju vremena odziva

Model	Sum of Squares	df	Mean Square	F	Sig.
Regression	$5{,}038 \cdot 10^{20}$	2	$2{,}519 \cdot 10^{20}$	115,294	***0,000***
Residual	$1{,}005 \cdot 10^{20}$	46	$2{,}185 \cdot 10^{18}$		
Total	$6{,}043 \cdot 10^{20}$	48			

Analize varijanse pokazuje da oba prediktora značajno utiču na vreme odziva, pošto je vrednost u koloni "Sig." 0.

Tabela 5.4 pokazuje koeficijente u linearnoj jednačini koja prikazuje uticaj nezavisnih promenljivih na vreme odziva.

Tabela 5.4 Koeficijenti u jednačini u modelu linearne regresije za predikciju vremena odziva

Model	Unstandardized Coefficients		Standardized Coefficients	t	Sig.
	B	Std. Error	Beta		
konstanta	$1{,}939 \cdot 10^{9}$	$4{,}965 \cdot 10^{8}$		3,906	***0,000***
broj korisnika	$6{,}221 \cdot 10^{7}$	$4{,}562 \cdot 10^{6}$	0,820	13,636	***0,000***
broj servera	$-1{,}254 \cdot 10^{9}$	$1{,}877 \cdot 10^{8}$	-0,402	-6,682	***0,000***

Iz tabele sledi da jednačina kojom se prikazuje zavisnost između srednjeg vremena odziva s jedne, i broja korisnika i broja aktivnih servera, s druge, može biti iskazana na sledeći način:

$$\bar{T} = 6.221 \cdot 10^{7} \cdot N_u - 1.254 \cdot 10^{9} \cdot N_s + 1.939 \cdot 10^{9}$$

odnosno:

$$\bar{T} = 0.820 \cdot N_u - 0.402 \cdot N_s$$

ako se koriste standardizovani koeficijenti.

U obe jednačine je T – očekivano vreme odziva u nanosekundama, N_u – broj korisnika koji pristupaju sistemu, a N_s – broj servera koji obrađuju klijentske zahteve.

Sve nezavisne promenljive su statistički značajni prediktori, pošto je za sve njih Sig=0.

Koeficijent N_s je negativan, što znači da sa porastom broja servera opada vreme odziva. Koeficijent N_s je pozitivan, jer vreme odziva raste sa brojem aktivnih korisnika. Kako je N_s oko 20 puta veći po apsolutnoj vrednosti od N_u, to znači da je potrebno dodati novi server na svakih 20 dodatnih klijenata da bi se vreme odziva održavalo na nekom očekivanom nivou.

Konstanta koja se pojavljuje u jednačini predstavlja uticaj ostalih faktora na sistem, kao što su: kašnjenje u mreži, vreme raspoređivanja zahteva u *node0* čvoru, itd.

Pored vremena odziva, od interesa je i broj, tj. procenat klijentskih zahteva koji biva odbačen zbog preopterećenosti servera. Analiza je sprovedena na isti način i dobijen je model čiji je rezime dat u tabeli 5.5.

Tabela 5.5 Rezime modela linearne regresije za predikciju broja neobrađenih klijentskih zahteva

R	R^2	Adjusted R Square	Standardizovana greška estimacije
0,841	***0,707***	0,694	0,029119

Izabrane nezavisne promenljive "objašnjavaju" 70.7% rezultata, što je statistički dobar rezultat.

Analiza varijanse je prikazana u tabeli 5.6.

Tabela 5.6 Analiza varijanse (ANOVA tabela) za model linearne regresije za predikciju broja neobrađenih klijentskih zahteva

Model	Sum of Squares	Df	Mean Square	F	Sig.
Regression	0,094	2	0,047	55,443	***0,000***
Residual	0,039	46	0,001		
Total	0,133	48			

Analiza varijanse pokazuje da su obe promenljive dobri prediktori, pošto je i ovde vrednost u koloni "Sig" 0.

Koeficijenti linearne zavisnosti su dati u sledećoj tabeli.

Tabela 5.7 Koeficijenti u jednačini u modelu linearne regresije za predikciju broja neobrađenih klijentskih zahteva

Model	Unstandardized Coefficients		Standardized Coefficients	t	Sig.
	B	Std. Error	Beta		
konstanta	0,032	0,010		3,323	0,002
broj korisnika	0,001	0,000	0,720	9,023	0,000
broj kompjutera	-0,020	0,004	-0,433	-5,429	0,000

Obe nezavisne promenljive su značajni prediktori (za obe Sig. ima vrednost 0), pa je dobijena jednačina:

$$\bar{P}_r = 0.001 \cdot N_u - 0.020 \cdot N_s + 0.032$$

odnosno:

$$\bar{P}_r = 0.720 \cdot N_u - 0.433 \cdot N_s$$

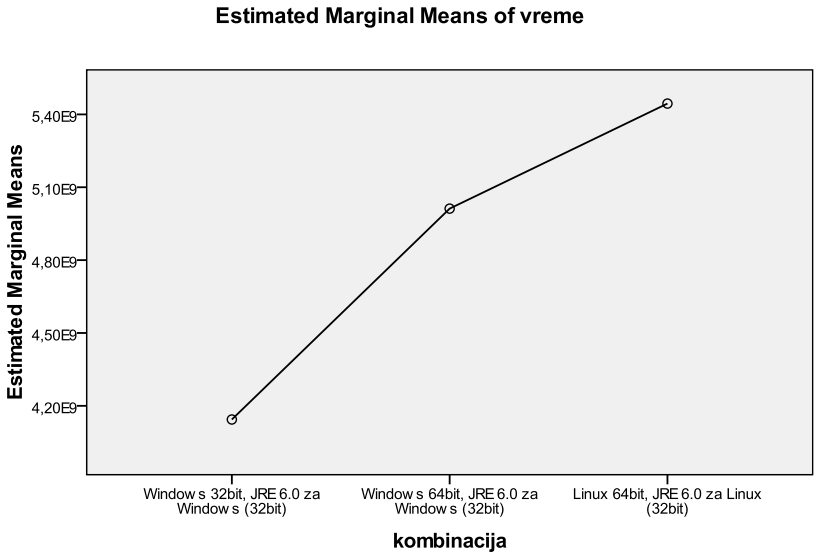
ako se koriste standardizovani koeficijenti.

I u ove dve jednačine je P_r – procenat neobrađenih klijentskih zahteva, N_u – broj korisnika koji pristupaju sistemu, a N_s – broj servera koji obrađuju klijentske zahteve.

Slično kao i kod vremena odziva, povećanje broja servera smanjuje procenat odbačenih zahteva. Povećanje broja klijenata povećava procenat odbačenih zahteva.

Slika 5.2 pokazuje da četiri servera, koji imaju slične hardverske karakteristike, pošto imaju različite softverske konfiguracije, daju različita vremena odziva.

Vidi se da je vreme odziva najmanje u konfiguracijama gde se koristi 32-bitni operativni sistem i 32-bitno Java okruženje. Odziv u konfiguracijama koje imaju 64-bitni operativni sistem i 32-bitno Java okruženje je sporiji, što je u skladu sa rezultatima koje su prijavili još neki korisnici ([PerfJava64a][PerfJava64b]).



Slika 5.2 Poređenje vremena odziva za različite konfiguracije

Dalja analiza može da pokaže i uticaj pojedinačnih računara na rezultate. Koristi se MANCOVA – multivarijantna analiza kovarijanse [Raykov08]. U postupku analize utvrđuje se efekat konfiguracije sistema na vreme odziva i broj neobrađenih zahteva, dok se broj korisnika koji pristupaju aplikaciji kontroliše, tj. eliminiše se uticaj broja korisnika na ponašanje sistema.

Deskriptivna statistika za ovu analizu data je u tabeli 5.8.

Tabela 5.8 Deskriptivna statistika u MANCOVA analizi efekta konfiguracija serverskih čvorova

	konfiguracija	**srednje vreme**	**standardna devijacija**
srednje vreme izvršavanja	Windows 32bit, JRE 6.0 za Windows (32bit)	$4{,}14 \cdot 10^9$	$3{,}464 \cdot 10^9$
	Windows 64bit, JRE 6.0 za Windows (32bit)	$5{,}01 \cdot 10^9$	$4{,}619 \cdot 10^9$
	Linux 64bit, JRE 6.0 za Linux (32bit)	$5{,}44 \cdot 10^9$	$5{,}049 \cdot 10^9$
	prosek	$4{,}69 \cdot 10^9$	$4{,}064 \cdot 10^9$
procenat neobrađenih zahteva	Windows 32bit, JRE 6.0 za Windows (32bit)	0,059	0,074
	Windows 64bit, JRE 6.0 za Windows (32bit)	0,068	0,050

	Linux 64bit, JRE 6.0 za Linux (32bit)	0,073	0,042
	prosek	0,065	0,060

Multivarijantni test je prikazan u tabeli 5.9.

Tabela 5.9 Rezultat multivarijantnog testa u MANCOVA analizi efekta konfiguracija serverskih čvorova

efekat	Wilks' Lambda	F	df1	df2	Sig.
konfiguracija	0,383	7,070	4	46	0,000
broj korisnika	0,012	961,021	2	23	0,000

Test pokazuje da je sastav konfiguracije značajan prediktor vremena odziva i procenta neobrađenih zahteva (Sig. je 0). Isto važi i za broj korisnika, što je pokazano u prethodnoj analizi.

Test pojedinačnih efekata je prikazan u tabeli 5.10.

Tabela 5.10 Pojedinačni efekti u MANCOVA analizi efekta konfiguracija serverskih čvorova

efekat	zavisna promenljiva	SS	df	MS	F	p
konfiguracija	vreme	$8{,}9 \cdot 10^{18}$	2	$4{,}44 \cdot 10^{18}$	6,43	0,01
	procenat neobrađenih zahteva	$1{,}1 \cdot 10^{-3}$	2	$5{,}33 \cdot 10^{-4}$	0,63	0,04
broj korisnika	vreme	$4{,}2 \cdot 10^{20}$	1	$4{,}20 \cdot 10^{20}$	607,98	0,00
	procenat neobrađenih zahteva	$7{,}7 \cdot 10^{-2}$	1	$7{,}65 \cdot 10^{-2}$	90,53	0,00

Konfiguracija objašnjava 96.3% procenata vremena odziva (R^2=0.963) i 76.7% procenta neobrađenih zahteva (R^2=0.767). p-test pokazuje da je softverska konfiguracija značajan prediktor vremena odziva (p=0.01) i procenata neobrađenih zahteva (p=0.04), jer je $p<0.05$ u oba slučaja.

5.2 Planiranje podmlađivanja softvera

Primer planiranja periodičnog podmlađivanja softvera biće prikazan na curenju memorije. Pojava curenja memorije predstavlja pojavu u kojoj

program vremenom, tokom izvšavanja, zauzima sve više memorije zato što ne oslobađa memoriju koju više ne koristi.

5.2.1 Curenje memorije u programskim jezicima C++ i Java

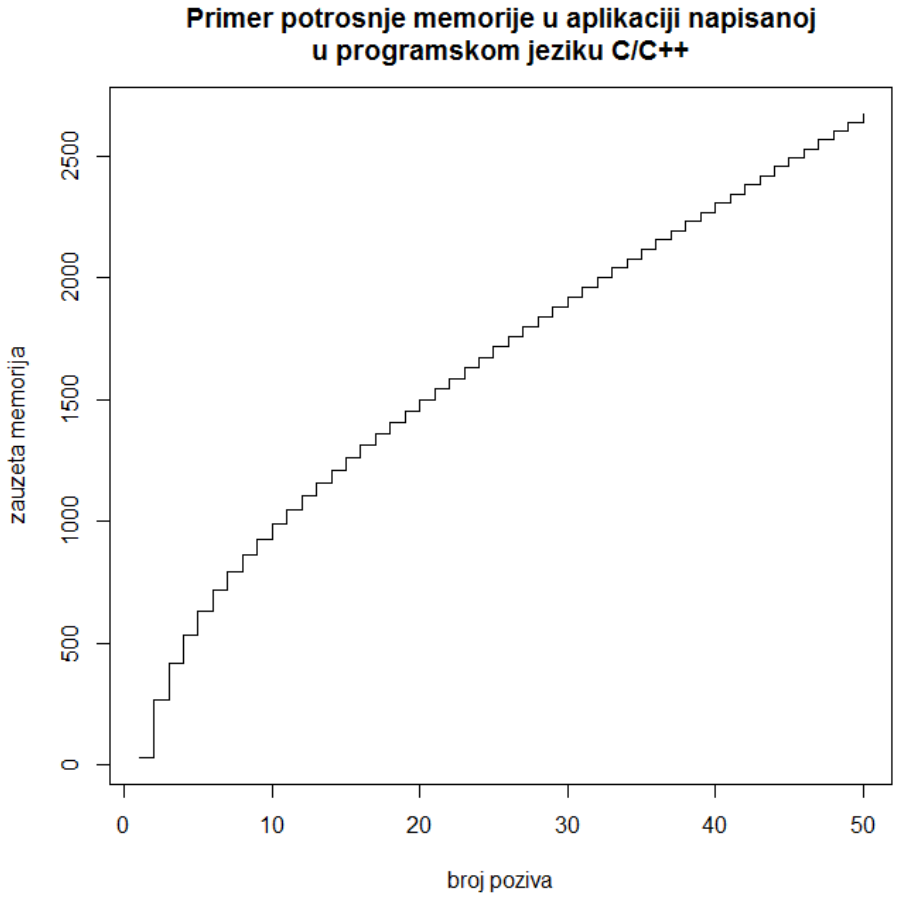
Curenje memorije se češće javlja u aplikacijama pisanim u C/C++, nego u programskom jeziku Java. U aplikacijama pisanim u C/C++ programskom jeziku, programer sam mora da vrši zauzimanje i oslobađanje memorije za objekte. Neka je dat kod kao u listingu u kom se zauzima memorija za dva niza od po deset karaktera.

```
1 {
2     char *memoryArea = malloc(100);
3     char *newArea = malloc(100);
4     memoryArea = newArea;
5     // do something
6     free(memoryArea);
7     free(newArea)
8 }
```

Listing 5.1 Primer koda koji izaziva curenje memorije u programskom jeziku C/C++

U drugoj i trećoj liniji koda zauzimaju se dva niza od po sto lokacija u memoriji. Nakon četvrte linije, pokazivač `memoryArea` počinje da pokazuje na lokacije na koje i `newArea`. Lokacije na koje je do tada pokazivao ostaju da "vise", tj. postaju nedostupne i zauzeta memorija ne može da bude oslobođena. Poziv `free(memoryArea)` (linija 6) oslobodiće samo lokacije koje su zauzete u liniji 3. Pri izvršavanju linije 7 doći će do greške, jer su lokacije oslobođene pri izvršavanju linije 6.

Uobičajeno ponašanje programa koji ima curenje je da se količina memorije koju zauzima vremenom povećava, obično u određenim koracima, kao na slici 5.3. Na početku izvršavanja obično dolazi do ubrzanog zauzimanja memorije (zbog pokretanja aplikacije i učitavanja većeg broja klasa), a kasnije porast zauzete memorije dobija linearan karakter, kao što je očekivano zbog curenja.



Slika 5.3 Primer curenja memorije tokom izvršavanja aplikacije napisane u programskom jeziku C/C++

U programskom jeziku Java ovo ne može da se dogodi. Java ima Garbage Collector, koji ima zadatak da sve objekte za koje je zauzeta memorija, a na koje ništa u programu ne pokazuje, obriše i oslobodi memoriju.

Curenje memorije u Javi može da se pojavi ako se trajno čuvaju reference na objekte koji se više ne koriste. Primer za ovo može da bude situacija u kojoj server u listi čuva identifikacije svih sesija, čak i onih koje su završene.

Neka je dat jednostavan primer u listingu .

```
1 // ...
2   ArrayList<Long> sessionIds = new
       ArrayList<Long>();
3   while(...){
4     String request = recieveClientRequest();
5     long sessionId = generateSessionId(request);
6     sessionIds.add(sessionId);
7     communicateWithClient();
8   }
9 // ...
```

Listing 5.2 Primer koda koji izaziva curenje memorije u programskom jeziku Java

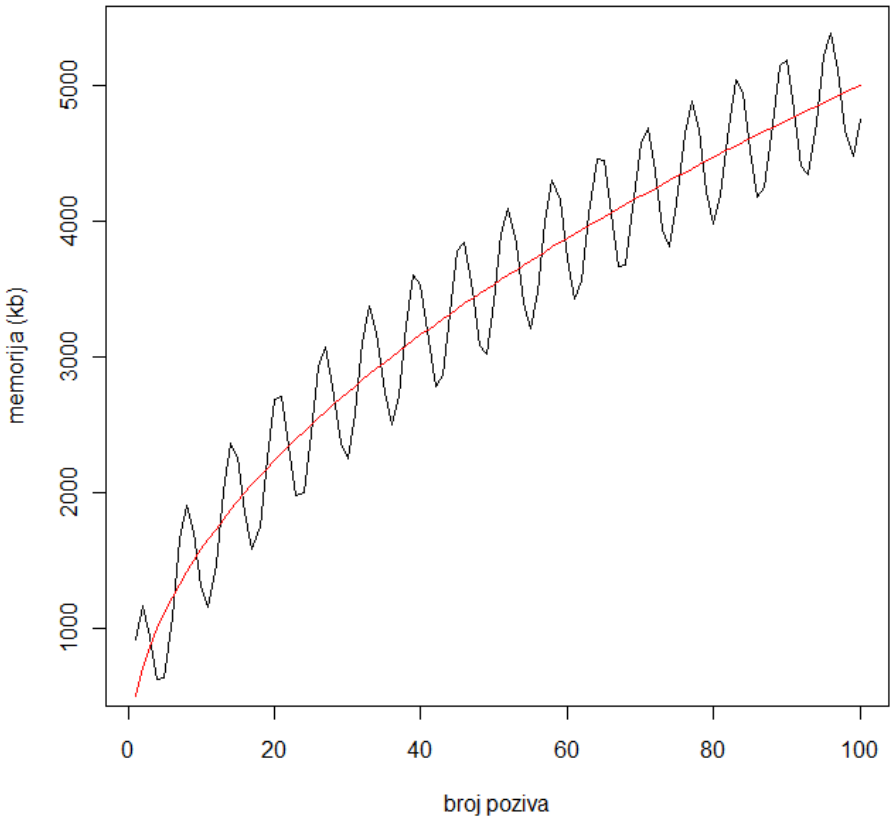
Svako izvršavanje ovog dela koda ubaciće novi `sessionId` u listu. Posle određenog vremena i broja klijenata, lista može da naraste previše i program može da prekorači rezervisanu količinu memorije. Ako bi nakon linije 7, bila ubačena linija sa listinga 5.3 problem bi bio rešen.

```
7 sessionIds.remove(sessionId);
```

Listing 5.3 Uklanjanje sesije koja je završena

Normalno za programe pisane u Java okruženju je da na početku zauzimaju veću količinu memorije, jer se u tom periodu obično vrši učitavanje klasa. Kasnije, kada program uđe u stabilan režim rada, količina memorije koju zauzima tokom vremena može da bude opisana sinusoidom. Program vremenom zauzima memoriju za objekte (rastuća ivica sinusoide), a Garbage Collector je periodično oslobađa (opadajuća ivica sinusoide).

Funkcija koja opisuje zauzeće memorije programa koji je dat u primeru bi bila slična funkciji sa slike 5.4.



Slika 5.4 Primer curenja memorije tokom izvršavanja aplikacije napisane u programskom jeziku Java

Funkcija sa slike odgovara funkciji koja je suma sinusoidalne i još jedne monotono rastuće funkcije. Sinusoida predstavlja normalno ponašanje programa (zauzimanje memorije za objekte i oslobađanje memorije od strane Garbage Collector-a). Druga, monotono rastuća, komponenta predstavlja nepredviđeno curenje memorije.

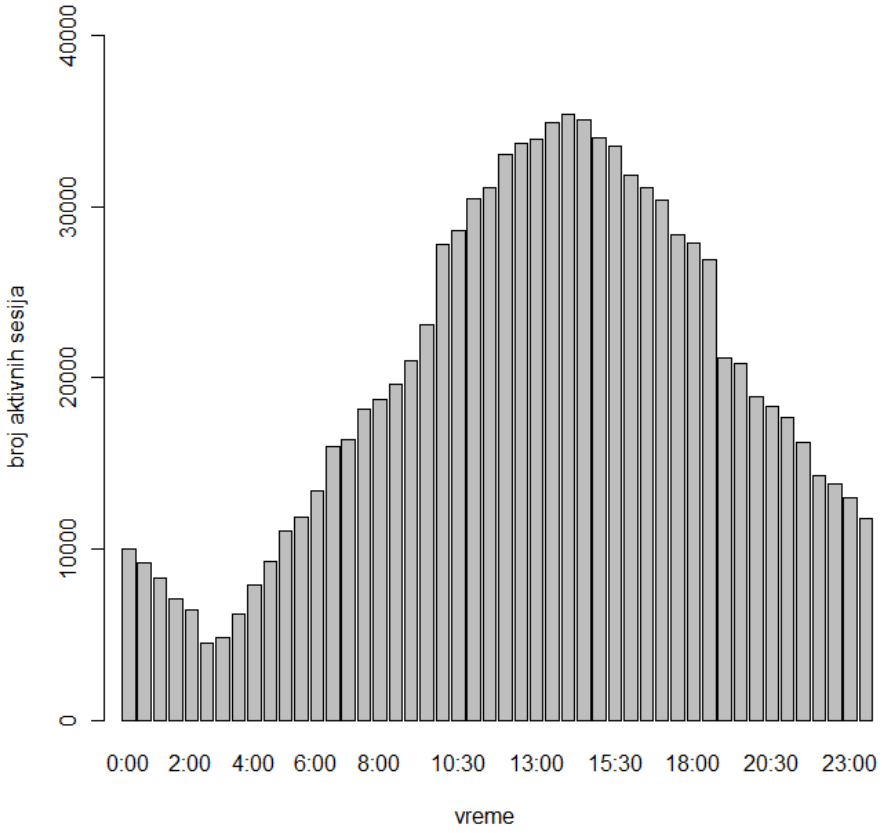
Na osnovu rezultata praćenja zauzeća memorije za neki softver, moguće je otkriti curenje memorije Java programa, a zatim predvideti posle kog vremena treba da se izvrši planirano podmlađivanje programa.

5.2.2 Planiranje podmlađivanja test aplikacije

Predviđanje će biti prikazano na aplikaciji iz poglavlja 4.1.1. U nju je ubačeno evidentiranje korisničkih poziva beleženjem sesije, na način kao što je to urađeno u listingu 5.2.

U prvih mesec dana rada aplikacije vršeno je praćenje aktivnosti korisnika da bi se videlo koje je vreme najpogodnije za redovno održavanje. Za praćenje je upotrebljen jedan aspekat koji beleži aktivne sesije korisnika.

Na osnovu rezultata, dobijen je sledeći grafik aktivnosti korisnika tokom perioda od 24h.



Slika 5.5 Primer curenja memorije tokom izvršavanja aplikacije napisane u programskom jeziku Java

Rezultati praćenja dalje pokazuju da je najmanje korisnika pristupa aplikaciji u periodu od 2 časa do 2:30 časova ujutru.

Tokom prvih mesec dana rada aplikacije utvrđeno je da aplikacija vremenom zauzima sve više memorije. Posle nekog vremena dešavao se prekid rada aplikacije zbog OutOfMemoryError izuzetka. Povećanje dozvoljene memorije za program pomoću JVM direktiva Xms i Xmx nije pomoglo. Intervencija je samo povećala vreme do novog prekida rada.

Ustanovljeno je da do pada dolazi kada se pređe broj od 1005000 poziva aplikacije (pri originalnim podešavanjima dozvoljene memorije).

Do pada aplikacije je dolazilo periodično, ali kako nije bilo moguće da se predvidi tačan trenutak kada će se tačno pad dogoditi, obično je bilo potrebno oko pola sata da se pad prijavi i da administratori reaguju. Za restartovanje aplikacije potrebno je pet minuta.

Analizom rezultata praćenja utvrđeno je da se nova sesija zabeleži u proseku svakih 0.3 sekunde. To znači da, pri datoj frekvenciji pristupanja aplikaciji od strane korisnika, aplikacija pada svakih 3.49 dana.

DProf je konfigurisan tako da traži koji deo aplikacije ima najveću potrošnju memorije na način na koji je opisano u poglavlju 3.5. Upotrebljen je aspekt sa listinga 4.4.

U narednom periodu, DProf-om je ustanovljeno da je do povećanja potrošnje memorije došlo pri beleženju klijentske sesije.

Da bi se ispravila ova greška, kao i još neke koje su otkrivene u međuvremenu, i nova verzija aplikacije istestirala i pustila u rad, razvojnom timu je potrebno mesec dana. U međuvremenu, periodično biće primenjeno planirano podmlađivanje aplikacije.

Odlučeno je da se podmlađivanje vrši na svaka tri dana, u periodu od 2 časa do 2:30 časova ujutru. Korisnici aplikacije su obavešteni o ovoj proceduri, tj. obavešteni su o datumima u narednih mesec dana kada aplikacija neće biti dostupna zbog održavanja.

Aplikacija zahvaljujući ovoj intervenciji nije dostupna samo u kraćem vremenskom periodu i samo u vremenu kada najmanje korisnika pristupa. Tako se povećala dostupnost aplikacije, dok se ne izvrši postavljanje nove verzije aplikacije.

6 Zaključak

Tema doktorske teze pripada oblasti praćenja rada softvera. Glavni rezultati prikazani u tezi su:

1. Predložen model DProf sistema koji može da vrši praćenje rada distribuiranih aplikacija. Model obuhvata sistem za praćenje i DProfSLA XML shemu. Na osnovu sheme se kreiraju XML dokumenti pomoću kojih se definišu očekivane performanse softvera koji se prati, kao i delovi aplikacije koji se prate.
2. Implementacija sistema izvršena je u JEE tehnologiji, uz oslonac na Kieker okruženje.
3. Sistem dodaje veoma malo opterećenje u okruženje u kom se izvršava, tako da predstavlja minimalnu smetnju radu softvera koji prati.
4. Izvršena je verifikacija praćenjem performansi aplikacije koja se izvršava na JBoss aplikativnom serveru.
5. Izvršeno je modelovanje zavisnosti performansi od uticaja okruženja u kom se izvršava - broja korisnika koji pristupaju aplikaciji i broja servera na kojima se aplikacija izvršava.
6. Prikazana je mogućnost primene dobijenih rezultata u planiranju održavanja aplikacije u situaciji kad aplikacija ima grešku koja može da dovede do prestanka rada.

U tezi je dat pregled radova koji se bave pitanjima performansi softvera. Pored radova koji se bave praćenjem performansi, prikazani su i standardi iz oblasti definisanja ugovora o nivou servisa, kao i radovi koji se bave predviđanjem promena performansi softvera.

Osnovne prednosti prikazanog DProf sistema u odnosu na postojeće sisteme su:

1. manje inicijalno opterećenje na posmatrani sistem - ovo je ostvareno arhitekturom sistema koja odlaže beleženje zapisa u bazu podataka do trenutka kada to najmanje utiče na rad krajnjih korisnika,
2. adaptivnost - prikupljanje podataka se uključuje samo u delovima sistema u kojima je detektovano postojanje problema, kao i
3. definisanje performansi pomoću XML-a.

DProf sistem je modelovan pomoću objedinjenog jezika za modelovanje (eng. *Unified Modeling Language* - UML). Sistem je implementiran u JEE tehnologiji, uz oslonac na Kieker okruženje. Kieker okruženje je prošireno dodatnim komponentama koje su opisane u ovoj tezi.

Komponenta ResultBuffer prihvata rezultate merenja i ne šalje ih odmah u bazu podataka, nego periodično. Na ovaj način se smanjuje opterećenje na sistem i uticaj na performanse, što je glavni problem svih alata za kontinuirano praćenje. Komponenta može da se konfiguriše tako da rezultate šalje u periodima kada je aktivnost korisnika aplikacije najmanja.

Pomoću Analyzer komponente vrši se analiza zapisa i po potrebi kreiraju novi parametri praćenja. Ovi parametri se šalju komponenti DProfManager. Ako Analyzer u rezultatima otkrije odstupanje od očekivanih vrednosti, on šalje DProfManager-u nove parametre za praćenje. Naime, sistem je inicijalno podešen da prati samo metode koje klijenti direktno pozivaju. Ako se detektuje odstupanje, pomoću novih parametara se uključuje praćenje metoda pozvanih iz onih prvih metoda, zatim metoda koje se pozivaju iz njih, itd. sve dok se ne nađe metoda koja je uzrok problema.

Standardni Kiekerov zapis koji predstavlja belešku o vremenu izvršavanja jedne metode je proširen. Novi tip zapisa je napravljen tako da prihvata i druge metrike. Merenje može da se vrši pomoću alata koji su standardno deo Java okruženja, ali i dodatnih (eng. *third party*) alata. Prikupljanje podataka se vrši pomoću aspekt-orijentisanog programiranja.

Definisana je i XML shema na osnovu koje se kreiraju dokumenti u kojima se zadaju očekivane vrednosti performansi u zadatim jedinicama. Ova shema je definisana u skladu sa važećim standardima u polju definisanja nivoa servisa.

Verifikacija modela je izvršena na reprezentativnoj JEE aplikaciji - aplikaciji koja se koristi za upravljanje softverskim konfiguracijama. Prvo je izvršeno merenje trajanja izvršavanja pojedinačnih metoda. U drugom primeru prikazano je kako DProf može da pronađe metodu tokom čijeg izvršavanja dolazi do povećane potrošnje memorije.

Na osnovu dobijenih rezultata pokazano je da DProf unosi manje opterećenje u sistem u odnosu na komponente koje su deo standardne Kieker distribucije.

U pretposlednjem poglavlju prikazano je kako dobijeni rezultati mogu da se primene za planiranje daljih akcija u održavanju softverskog sistema. U prvom primeru je prikazana primena linearne regresije nad rezultatima da bi predvideli dalje akcije u cilju održavanja nivoa servisa - tj. pronađena je zavisnost na osnovu koje može da se proceni kada je potrebno u sistem dodati novi server da bi se odgovorilo na povećane zahteve sve većeg broja korisnika. Drugi primer prikazuje primenu dobijenih rezultata na planiranje preventivnog restartovanja aplikacije. Ovaj postupak se koristi u situacijama kad aplikacija ima grešku zbog koje ne oslobađa resurse koje više ne koristi, pa može dođe u situaciju da potroši resurse (npr. "curenje"

memorije, konekcija prema bazi podataka). Ako greška ne može odmah da se ukloni, planira se preventivno gašenje i pokretanje aplikacije, nakon čega ona oslobađa resurse i može normalno da radi.

Nedostatak sistema je taj što nakon izmene parametara praćenja, posmatrani softver mora da se restartuje. Ovo izaziva gubljenje korisničkih sesija i predstavlja problem posebnih istraživanja. Jedan od načina da se ovo reši je da se koriste neka od okruženja koja dozvoljavaju izmenu aspekata upredenih sa klasama tokom rada aplikacije. Ova okruženja postoje, ali su u eksperimentalnoj fazi.

Pored rešavanja problema gubitka sesije, dalji razvoj sistema je usmeren na čvršću integraciju sa Kieker okruženjem. Ovo se prvenstveno odnosi na razvoj Analyzer komponente, koja bi trebalo da bude razvijena uz oslonac na Kieker.Analysis komponentu. Ovo nije urađeno odmah jer je Kieker.Analysis komponenta više puta menjala arhitekturu tokom razvoja DProf sistema.

Summary

Constant growth in the size and complexity of modern enterprise applications makes them extremely demanding both from a functional and non-functional aspect. Non-functional requirements, such as availability, responsiveness, robustness, portability, etc., are defined in an agreement between software providers and consumers called “service level agreement” (SLA) [Meyer05]. Because of the pressure to put the application into operation, there is usually very little time for testing and bug checking. Also, debuggers and profilers hardly allow detecting all errors and unpredicted events that occur during production and operation. It is a very common phenomenon that software performance and “quality of service” (QoS) degrade over time [Grottke08]. This calls for continuous monitoring of applications in order to determine whether QoS is kept on a satisfactory level. Continuous monitoring provides a picture of dynamic software behavior under real exploitation circumstances. The data obtained from monitoring can for instance be used as a basis for architecture-based software optimization, visualization, and reconstruction [Hoorn12]. It can also be applied to software performance prediction and maintenance planning [Okanovic12c][Okanovic12d].

Performance overhead that a monitoring system imposes is a very important issue. The monitoring system has to work using a minimal amount of resources. Profilers and debuggers induce significant performance overhead and are therefore unsuitable for monitoring during the operational phase. In order to achieve a reduction of monitoring overhead, it would be beneficial to automatically adapt monitoring to only monitor selected parts of the system.

Based on the data collected using continuous monitoring, it is possible to predict the behavior of the application and make a plan of further actions. This is, for example, important ,in capacity planning and maintenance scheduling.

The DProf system proposed in this thesis has been developed for adaptive monitoring of distributed enterprise applications with a low overhead. In order to do that, the Kieker [Hoorn12] framework, which yields low overhead, is used for collecting the monitoring data. Additional components support the change of monitoring parameters at application runtime. These additional components have been developed using Java Management Extensions (JMX) [JMX]. The system analyzes call trees reconstructed from the gathered data and automatically creates a new monitoring configuration if needed. A call tree represents calling

relationships between software methods [Binder06]. It contains the control-flow of method executions invoked by a client request. The first method is called the "root".

DProf configuration parameters specify which of the application's call trees are going to be monitored and can furthermore specify nesting levels within the call tree that will be monitored. DProf stores data in a central database, regardless of on how many computers the monitored application is being executed. Using a mechanism integrated into the Kieker framework, during data gathering, each method execution within a trace is uniquely identified and is assigned a number which represents the order of execution. This allows call trees to be spread on different computers.

DProf reduces monitoring overhead by only monitoring parts of the software suspected of containing problems or deviating from expected behavior. The system starts by monitoring methods that are at the root of the call trees. If a deviation from expected results is detected in one of the trees, the DProf incrementally turns on monitoring in lower levels of that particular tree until the method that is causing the problem is determined. DProf adapts without human intervention to find the cause of the problem. This simulates the manual procedure typically employed for localizing the root cause of performance problems. Other systems perform monitoring of the whole software, regardless of the fact that other parts (i.e. other call trees) are working fine. Since DProf's additional monitoring components are implemented using JMX technology, the reconfiguration of the DProf monitoring parameters can still be performed manually by system administrators using any JMX console.

The first chapter introduces basic concepts in the field of performance monitoring and performance prediction. The importance of SLA monitoring is highlighted as it assures data needed for service level management. The behavior of software containing bugs, and the correlation between these bugs and execution environment is explained. The chapter also shows an overview of dynamic and static software analysis techniques. It explains the advantage of continuous monitoring of service levels and software performance over monitoring tools used in the software testing phase.

The second chapter provides an overview of related work from this field. After an introduction that shows the development of monitoring tools, common tools used for Java applications monitoring are presented. This overview shows that the use of AOP [Kiczales96] yields lower overhead than JVMTI/JVMPI [JVMTI][JVMPI] and therefore AOP based tools are better for continuous monitoring. Very few of these tools have automatic adaptation functionality. One of them [Ehlers11] is based on the Kieker

framework, as DProf is, but it requires the knowledge of OCL [OCL]. The last part of this section analyses performance overhead that these tools imposed on the monitored system. Existing SLA document specification principles are discussed in the second part of this chapter. The SLA lifecycle is shown [Sturm00]. The paper by Paschke [Paschke06] provides categorization schemes for SLA documents. For this thesis, XML based SLA documents are of particular interest because XML is a machine readable format. As such, XML can be used for defining DProf adaptation rules. The third part of the chapter focuses on related work on the topic of performance prediction. Performance prediction is an important part of performance management, as it provides means for better maintenance and capacity planning, lowering the total cost of the operation. Most researchers in this field focus on creating the general model for performance prediction. In most cases this proved to be futile. Instead, it is better to create models for specific situations.

In the third chapter, specification and implementation details of DProf are presented. The system is divided into two parts. One part of the system monitors application data based on monitoring rules. This data is sent to the other part of the system which analyses obtained data. Based on this analysis, new monitoring configuration is created and sent back. Monitoring reconfigures using new parameters and resumes without the need for human interference. Monitoring is performed using the Kieker framework with the addition of a new writer and several new components. Unlike other Kieker writers, this writer stores monitoring data into a buffer and sends it periodically to the server. This results in lower monitoring overhead. Monitoring rules are defined using SLA documents which are created according to the new XML schema. These documents are much simpler than those presented in related work, and can easily be incorporated into them. The next section of this chapter presents the Kieker framework. The specification of the system is given in UML diagrams. It is implemented using JEE technology. AOP is used for the implementation of monitoring probes that perform measurements. The system can be used for the monitoring of software written for other platforms - only the monitoring part of the system needs to be implemented, while the analyzer part is platform independent. There is already a Kieker implementation in the .NET technology, thus, in this case, only few components have to be implemented.

The fourth chapter of the thesis shows how the DProf system can be used for monitoring a SCM application that is implemented using JEE technology. The application has been deployed on a cluster of servers. The monitoring scenarios consisted of simple application monitoring and

adaptive monitoring. In the first scenario, only gathering of monitoring data has been performed. This scenario should be used, for example, to determine normal values of the SLA. In the second scenario, the system was configured to locate the method that causes deviation from values defined in the SLA. One method in the monitored application was intentionally modified to generate a deviation from expected results. The subsequent analysis shows that DProf system yields lower overhead than the use of other Kieker writers for monitoring data gathering.

In the fifth chapter performance prediction using gathered monitoring data is shown. In the first example, a model was created by performing linear regression on the gathered monitoring data. This model allows the prediction of future performance depending on the number of users and servers on which the application is deployed. Another model was created to show how gathered data can be used to successfully plan and implement periodical application rejuvenation [Siewiorek92].

The last chapter provides conclusions and draws guidelines for future work. The main scientific contribution of this thesis is the specification and implementation of the DProf system. This system can be used for continuous application monitoring. It imposes a very small overhead compared to similar systems. This is partly because of its design, but mostly because the DProf adapts to monitor only parts of the applications that deviate from the designated values. Monitoring configuration is provided using a XML document that is based on the DProfSLA schema. The specification is given using UML and the system is implemented using JEE. It is also shown how the system can be used to monitor applications developed for other platforms. The thesis also shows how performance prediction can be used in capacity and maintenance planing and in performance prediction. The presented system was verified on a representative JEE application deployed on the cluster of JBoss servers.

Literatura

[Agarwal04] Agarwal, M., Appleby, K., Gupta, M., Kar, G., Neogi, A., Sailer, A., Sahai, A., Wu, F.: Problem Determination Using Dependency Graphs and Run-Time Behavior Models. Lecture Notes in Computer Science, vol. 3278, Springer Berlin / Heidelberg, p. 171-182 (2004)

[Aguilera03] Aguilera, Mogul, J., Wiener, J., Reynolds, P., Muthitacharoen, A.: Performance Debugging for Distributed Systems of Black Boxes. In Proceedings of the 19th ACM symposium on Operating systems principles. ACM, Bolton Landing, New York, USA. 74-89. 2003.

[Alonso10] Alonso, J., Torres, J., Berral, J.L., Gavaldà, R.: Adaptive On-line Software Aging Prediction Based on Machine Learning. IEEE International Conference on Dependable Systems and Networks. p. 507-516. (2010)

[Amdahl67] Amdahl, G.: Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring)). ACM, New York, USA. p. 483-485. (1967)

[Ammons97] G. Ammons, T. Ball, J. R. Larus, Exploiting hardware performance counters with flow and context sensitive profiling, Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, p.85-96, 1997, Las Vegas, USA

[Andrzejak06] Andrzejak, A., Silva, L.: Deterministic Models of Software Aging and Optimal Rejuvenation Schedules. CoreGRID TR-0047. (2006)

[ApacheServer] Apache HTTP Server Project. Apache Software Foundation. [Online] http://httpd.apache.org/

[APDT] APDT: Aspect PHP Developement Tools, http://code.google.com/p/apdt/

[Aquarium] Aquarium, Aspect-Oriented Programming for Ruby, http://aquarium.rubyforge.org/

[ASF] Apache Software Foundation, http://www.apache.org/

[ASM] ASM, http://asm.ow2.org/

[AppDynamics] AppDynamics. [Online] Available: http://www.appdynamics.com (2012)

[AspectDNG] AspectDNG, http://sourceforge.net/projects/aspectdng/

[AspectJ] AspectJ, http://www.eclipse.org/aspectj/

[AspectW] AspectWerkz, http://aspectwerkz.codehaus.org/

[Avižienis04] Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing,

IEEE Tranactions on Dependable and Secure Computing, v.1, n.1, p. 11-33, 2004.

[Bao05] Bao, Y., Sun, X., Trivedi, K.:A Workload-based Analysis of Software Aging and Rejuvenation. IEEE Transactions on Reliability, v. 54, n. 3, p. 541-548. (2005)

[BCEL] Byte Code Engineering Library. Apache Software Foundation. [Online] http://jakarta.apache.org/bcel/index.html

[Binder06] Binder, W., Portable and Accurate Sampling Profiling for Java, Software – Practice & Experience, v.36 n.6, p.615-650, 2006.

[Binder07] Binder, W., Hulaas, J., Moret, P.: Advanced Java Bytecode Instrumentation. Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, Lisboa, Portugal. p. 135-144 (2007)

[Bubak03] Bubak, M., Funika, W., Wismuller, R., Metel, P., Orlowski, R.: Monitoring of Distributed Java Applications. Future Generation Computer Systems, Elsevier. v.19 n.5. p.651-663 (2003)

[CAUnicenter] Application Performance Management. CA Technologies. [Online] Available: http://www.ca.com/us/application-performance-management.aspx (2012)

[Chen02] Chen, M., Kiciman, E., Fratkin, E.., Fox, A., Brewer, E.: Pinpoint: Problem Determination in Large Dynamic Systems. In Proceedings of 2002. International Conference on Dependable Systems and Networks. IEEE Computer Society, Washington DC, USA. 595-604. (2002)

[Clifford08] Clifford, D., van Bon, J.: Implementing ISO/IEC 20000 Certification: The Roadmap, ITSM Library, Van Haren Publishing, 2008. ISBN 908753082X

[COBOLAOP] Shinomi, H., Ichimori, Y.: Program Analysis Environment for Writing COBOL Aspects. Proceedings of the 9th International Conference on Aspect-Oriented Software Development, Rennes and Saint-Malo, France. p. 222-230 (2010)

[Cousot77] Cousot, P., Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. 4th Symposium on Principles of Programming Languages. p. 238-252 (1977)

[Cronk08] Cronk, B.: How to Use SPSS: A Step-By-Step Guide to Analysis and Interpretation. Pyrczak Publishing, USA.ISBN 978-1884585791 (2008)

[CSharpAOP] Schult, W., Polze, A.: Aspect-Oriented Programming with C# and .NET, Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. p. 241-249. (2002)

[dotspect] dotspect. [Online] http://dotspect.tigris.org/

[Dimitriev03] M. Dimitriev, Design of JFluid, Technical Report: SERIES13103, Sun Microsystems Inc., USA (2003)

[dynaTrace] dynaTrace – Continuous application performance management. dynaTrace software Inc. [Online] Available: http://www.dynatrace.com/ (2012)

[Ehlers11] Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-Adaptive Software System Monitoring for Performance Anomaly Localization. In Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011). ACM, Karlsruhe, Germany. 197-200. (2011)

[Ehmke12] Kieker 1.5 User Guide. Ehmke, N. C., Hoorn, A. v., Jung, R. (2012) [online] Available: http://netcologne.dl.sourceforge.net/project/kieker/kieker/kieker-1.5/kieker-1.5_userguide.pdf

[EJB3] EJB 3.0. Oracle. [Online] http://java.sun.com/products/ejb/ (April 2012)

[Emerson81] Emerson, E. A., Clarke, E. M.: Characterizing Correctness Properties of Parallel Programs Using Fixpoints. Proceedings of the 7th Colloquium on Automata, Languages and Programming. Springer-Verlag, London, UK. p. 169-181. (1981)

[Eustace95] Eustace, A., Srivastava, A.: ATOM: a Flexible Interface for building high performance program analysis tools. In Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings, p.25-25, January 16-20, 1995, New Orleans, Louisiana

[Gamma94] Gamma, E., Helm, R., Johnson, R., Vlissides, J. M: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Proffesional, Boston, USA. (1994)

[Gatlin05] Gatlin, K. S., Profile-Guided Optimization with Microsoft Visual C++, MSDN Library, 2005, http://msdn.microsoft.com/en-us/library/aa289170.aspx

[GNUBinutils] GNU Binutils, http://www.gnu.org/software/binutils/

[GNUGCC] GCC, the GNU Compiler Collection, http://gcc.gnu.org/

[GNUOS] GNU Operating System, http://www.gnu.org/software/binutils/

[Govindraj06] Govindraj, K., Narayanan, S., Thomas, B., Nair, P., Peeru, S.: On using AOP for Application Performance Management. In Industry Track Proceedings of the 5th International Conference on Aspect-Oriented Software Development. ACM, Bonn, Germany. (2006)

[Graham82] Graham, S., Kessler, P., Mckusick, M., Gprof: A Call Graph Execution Profiler. Proceedings of the 1982 SIGPLAN symposium on Compiler construction, Boston, United States, pg. 120 - 126 (1982)

[Grottke06] Grottke, M.: Analysis of Software Aging in a Web Server. IEEE Transactions on Reliability, v. 55, n. 3. p. 411-420. (2006)

[Grottke07] Grottke. M., Trivedi, K. S., Fighting Bugs: Remove, Retry, Replicate, Rejuvenate, IEEE Computer, v.40, n. 2, p. 107-109, 2007.

[Grottke08] Grottke, M., Matias, R., Trivedi, K.: The Fundamentals of Software Aging. Proceedings of 1st International Workshop on Software Aging and Rejuvenation/19th International Symposium on Software Reliability Engineering. p. 1-6. (2008)

[Hecht77] Hecht, M. S.: Flow Analysis of Computer Programs. Elsevier North-Holland Inc. (1977)

[Hoare69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, n. 10, p. 576–580, 583. (1969)

[Holzmann07] Holzmann, G. J., Conquering Complexity, IEEE Computer, v. 40, n. 12, p. 111-113, 2007.

[Hoorn12] Hoorn, A. v., Hasselbring, W., Waller, J.: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012). ACM, Boston, Massachusetts, USA. To appear. (2012)

[HotSpot] Java SE HotSpot at a Glance, http://java.sun.com/javase/technologies/hotspot/index.jsp

[HPInsight] HP Systems Insight Manager. Hewlett-Packard. [Online] Available: http://h18013.www1.hp.com/products/servers/management/hpsim/index.html?jumpid=go/hpsim (2012)

[HPROF] Using the HPROF Profiler. IBM (2010) [Online] http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/index.jsp?topic=/com.ibm.java.doc.diagnostics.50/diag/tools/jvmti.html

[Huang95] Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D. Software rejuvenation: Analysis, module and applications, Proceedings of 25th International Symposium on Fault Tolerant Computing (FTCS 25), pp. 381-390, 1995.

[HUTN] Human Usable Textual Notation (HUTN) Specification. OMG. [Online] Available: http://www.omg.org/spec/HUTN/index.htm (current September 2011)

[IBMDK] IBM Developer Kits. IBM (2010) [Online] https://www.ibm.com/developerworks/java/jdk/

[IBMTivoli] IBM - Monitoring Software - Tivoli Monitoring. IBM. [Online] http://www-01.ibm.com/software/tivoli/products/monitor/ (2012)

[Inoue09] Inoue, H., Nakatani, T., How a Java VM can get more from a hardware performance monitor, Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, 137-154, Orlando, USA, 2009.

[Ipek05] Ipek, E., de Supinski, B.R., Schulz, M., McKee, S.A.: An Approach to Performance Prediction for Parallel Applications,

Lecture Notes in Computer Science, vol. 3648/2005. p. 627-628. (2005)

[Jakarta] Jakarta, http://jakarta.apache.org/

[Jarvis06a] Jarvis, S. A., Spooner, D. P., Lim Choi Keung, H. N., Cao, J., Saini, S., Nudd, G. R.: Performance Prediction and Its Use in Parallel and Distributed Computing Systems. Future Generation Computer Systems – Systems Performance Analysis and Evaluation, v. 22, n. 7. Elsevier Science Publishers, Amsterdam, Netherlands. p. 745-754. (2006)

[Jarvis06b] Jarvis, S., Spooner, D., H.N.L.M. Keung, Cao, J., Saini, S., Nudd, G.: Performance Prediction and Its Use in Parallel and Distributed Computing Systems. Future Generation Computer Systems, v. 22, 745–754 (2006)

[JavaEE] Java EE at a Glance, http://java.sun.com/javaee/index.jsp

[Javassist] Javassist, http://labs.jboss.com/javassist/

[JBoss] JBoss, http://www.jboss.org/

[JBossAS] JBoss Application Server 5.0. Red Hat. [Online] http://www.jboss.org/jbossas

[JBossDS] Configuring Datasources. JBoss.org Community Documentation. http://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/4/html/Connectors_on_JBoss-Configuring_JDBC_DataSources.html

[JMX] Java Management Extensions Technology. Oracle. [Online] http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[JPDA] Java Platform Debugger Architecture. Oracle. [Online] http://java.sun.com/javase/technologies/core/toolsapis/jpda/index.jsp

[JSR163] JSR 163: JavaTM Platform Profiling Architecture, http://jcp.org/en/jsr/detail?id=163

[JVMPI] Java Virtual Machine Profiler Interface (JVMPI). Oracle. [Online] http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html (current April 2012)

[JVMTI] Java Virtual Machine Tool Interface (JVMTI). Oracle. [Online] http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/ (current April 2012)

[JXInsight] JXInsight. JInspired. [Online] Available: http://www.jinspired.com/products/jxinsight/ (2012)

[Kalin09] Kalin, M.: Java Web Services: Up and Running. O'Reilly Media, Sebastopol, California, USA. (2009)

[Kanmani04] Kanmani, S., Uthariaraj, R., Sankaranarayanan, V., Thambidurai, P.: Object Oriented Software Quality Prediction Using General Regression Neural Networks. ACM

SIGSOFT Software Engineering Notes, v. 29, n. 5. p. 1-6. (2004)

[Karunanithi92] Karunanithi, N., Whitley, D., Malaiya, Y.K.: Using Neural Networks in Reliability Prediction. IEEE Software, v. 9, n. 4. p. 53-59, 1992.

[Katchabaw97] Katchabaw, M., Howard, S., Lutfiyya, H., Marshall, A., Bauer, M.: Making Distributed Applications Manageable Through Instrumentation. Journal of Systems and Software, v.45, n.2. Elsevier Science, New York, USA. p. 81-97. (1997)

[Kalin09] Kalin, M.: Java Web Services: Up and Running. O'Reilly Media, USA, 2009.

[Keller03] Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management, v. 11 n. 1, p. 57-81. (2003)

[Kiczales96] Kiczales, G.: Aspect-Oriented Programming. ACM Computing Surveys (CSUR), v.28 n.4. (1996)

[Lafferty03] Lafferty, D., Cahill, V.: Language-Independent Aspect-Oriented Programming. ACM SIGPLAN Notices, v.38 n.11. (2003)

[Lamanna03] Lamanna, D., Skene, J., Emmerich, W.: SLAng: A Language for Defining Service Level Agreements. In Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '03). IEEE Computer Society, Washington, DC, USA. p. 100-107. (2003)

[Lambert08] Lambert, J., Power, J.: Platform Independent Timing of Java Virtual Machine Bytecode Instructions. Electronic Notes in Theoretical Computer Science (ENTCS), v. 220, p. 97-113, 2008.

[Larus94] Larus, J. R., Ball, T.: Rewriting Executable Files to Measure Program Behavior, Software—Practice & Experience, v.24 n.2, p.197-218, 1994.

[Liu04] Liu, R., Gibbs, C., Coady, Y.: MADAPT: Managed Aspects for Dynamic Adaptation Based on Profiling Techniques. In Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, Toronto, Ontario, Canada. p. 214 - 219. (2004)

[Loo11] extremevalues: Univariate Outlier Detection. Mark van der Loo. (2011) [Online] Available: http://cran.r-project.org/web/packages/extremevalues/ (current September 2011)

[Loom] Welcome to the LOOM.NET Project!. Operating Systems and Middleware Group at HPI. [Online] http://www.dcl.hpi.uni-potsdam.de/research/loom/

[Ludwig97] Ludwig, T., Wismuller, R., Sunderam, V., Bode, A.: OMIS 2.0 - On-line Monitoring Interface Specification (Version 2.0). Technishe Universitat Munchen. Munich, Germany. TUM-

I9733. (1997) [Online] http://www.lrr.in.tum.de/~omis/OMIS/Version-2.0/version-2.0/

[Marwede09] Marwede, N., Rohr, M., van Hoorn, A., Hasselbring, W.: Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems Based on Timing Behavior Anomaly Correlation. In Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09). IEEE Computer Society, Kaiserslautern, Germany. 47-58. (2009)

[Meyer05] Meyer, D.: Beneath the Buzz: ITIL. CIO Magazine. (2005) [Online] http://web.archive.org/web/20050404165524/www.cio.com/leadership/buzz/column.html?ID=4186

[MOF] Meta Object Facility (MOF) 2.0 Core Specification. OMG. [Online] Available: http://www.omg.org/spec/MOF/2.0 (current September 2011)

[modjk] The Apache Tomcat Connector. Apache Software Foundation. [Online] http://tomcat.apache.org/connectors-doc/

[Nagios] Nagios. [Online] Available: http://www.nagios.org (2012)

[NetBeans] NetBeans, http://netbeans.org/index.html

[Nudd00] Nudd, G. R., Kerbyson, D. J., Papaefstathiou, E., Perry, S. C., Harper, J. S., Wilcox, D. V.: Pace – A Toolset for the Performance Prediction of Parallel and Distributed System. The International Journal of High Performance Computing Applications, v. 14, n. 3. p. 228-251. (2000)

[OCL] Object Constraint Language (OCL) 2.0. OMG. [Online] Available: http://www.omg.org/spec/MOF/2.0 (Septembar 2011)

[OHair04] O'Hair, K. The JVMPI Transition to JVMTI. Sun Developer Network (2004) [Online] http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/

[Okanovic06] Okanović, D., "Sistem za praćenje rada aplikacija baziranih na JMX tehnologiji", magistarski rad, Fakultet tehničkih nauka, Novi Sad, 2006.

[Okanovic08] Okanović, D., Vidaković, M.: One Implementation of the System for Application Version Tracking and Automatic Updating. Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria. p. 80-85. (2008)

[Okanovic10] Okanović, D., Vidaković, M., Kovačević, A.: Integracija Kieker sistema i JBoss-a. YuInfo 2010. Kopaonik, Srbija. CD. (2010)

[Okanovic11a] Okanović, D., Vidaković, M.: Performance Profiling of Java Enterprise Applications. In Proceedings of the 1st International Conference on Information Society Technology and Management. Kopaonik, Serbia. p. 60-64. (2011)

[Okanovic11b] Okanović, D., Hoorn, A. v., Konjović, Z., Vidaković, M.: Towards Adaptive Monitoring of Java EE Applications. In Proceedings of the 5th International Conference on Information Technology - ICIT. Amman, Jordan. CD. (2011)

[Okanovic11c] Okanović, D., Konjović, Z., Vidaković, M.: Continuous Monitoring System for Software Quality Assurance. In Proceedings of the 15th International Scientific Conference on Industrial Systems (IS11). Novi Sad, Srbija. CD. (2011)

[Okanovic12a] Okanović, D., Vidaković, M., Konjović, Z.: Service Level Agreement XML Schema for Software Quality Assurance. Acta Technica Corviniensis, vol. 5, n. 1. p. 123-128. (2012)

[Okanovic12b] Okanović, D., Konjović, Z., Vidaković, M.: Low-Overhead Continuous Monitoring of Service Level Agreements. International Journal of Industrial Engineering and Management (IJIEM), v.3, n.4. p. 25-31. (2012)

[Okanovic12c] Okanović, D., Vidaković, M.: Software Performance Prediction Using Linear Regression. In Proceedings of the 2nd International Conference on Information Society Technology and Management. Kopaonik, Serbia. p. 60-64. (2012)

[Okanovic12d] Okanović, D., Vidaković, M., Konjović, Z.: Monitoring of JEE Applications and Performance Prediction. Journal of Information Technology and Applications, v.1, n.2. p. 136-143. (2012)

[Oldham06] Oldham, N., Verma, K., Sheth, A., Hakimpour, F.: Semantic WS-Agreement Partner Selection. In Proceedings of the 15th international conference on World Wide Web. ACM, New York, New York, USA. 697-706. (2006)

[OW2] OW2 Consortium, http://www.ow2.org/

[Parnas96] Parnas, D. L., Software Aging, Procedings of 16th International Conference on Software Engineering, p. 279-287, Sorrento, Italy, 1994.

[Paschke06] Paschke, A., Schnappinger-Gerull, E.: A Categorization Scheme for SLA Metrics. In Proceedings of Multi-Conference Information Systems (MKWI 2006), Passau, Germany. (2006)

[Pearce07] Pearce, D., Webster, M., Berry, R., Kelly, P.: Profiling with AspectJ. Software—Practice & Experience, v.37 n.7, p.747-777. (2007)

[PerfJava64a] Java Performance 64-bit. StackOverflow. [Online] http://stackoverflow.com/questions/4934297/java-performance-64-bit

[PerfJava64b] Microsoft Answers - Java 32bit Kills Win 7 Performance. Microsoft. [Online] http://answers.microsoft.com/en-us/windows/forum/windows_7-performance/java-32bit-kills-win-7-performance/d41ed278-0e26-4fb5-a18c-43900867fea0

[Pixie89] RISCompiler and C Programmer's Guide. MIPS Computer Systems Inc. (1989)

[Pooley2000] Pooley, R., Software engineering and performance: a roadmap, International Conference on Software Engineering, Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland (2000)

[Prof79] Unix Programmer's Manual, prof Command, section 1. Bell Laboratories, Murray Hill, USA. (1979)

[Raykov08] Raykov, T.: An Introduction to Applied Multivariate Analysis. Routledge Academic, USA. ISBN 978-0805863758 (2008)

[Reiss08] Reiss, S.: Controlled Dynamic Performance Analysis. In Proceedings of the 7th international workshop on Software and performance. Princeton, USA. (2008)

[RLanguage10] R Development Core Team. R: A language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. (2010)

[Rohr08] Rohr, M, Hoorn, A. v., Matevska, J., Sommer, N., Stoever, L., Giesecke, S., Hasselbring, W.:Kieker: Continuous Monitoring and on Demand Visualization of Java Software Behavior. In Proceedings of the IASTED International Conference on Software Engineering (SE '08). ACTA Press, Anaheim, CA, USA. p. 80-85. (2008)

[Roubtsov03] Roubtsov, V.: Sizeof for Java [Online] http://www.javaworld.com/javaworld/javaqa/2003-12/02-qa-1226-sizeof.html (2003)

[Rutherford06] Rutherford, A.: Introducing Anova and Ancova: A GLM Approach. Sage Publications, USA, ISBN 978-0761951612 (2006)

[Schade96] Schade, A., Trommler, P., Kaiserswerth, M.: Object Instrumentation for Distributed Applications Management. Proceedings of the IFIP/IEEE International Conference on Distributed Platforms (ICDP'96), Dresden, Germany. IFIP, Chapman and Hall. (1996)

[SIGAR] SIGAR API - System Information Gatherer and Reporter. Hyperic. [Online] http://www.hyperic.com/products/sigar

[Siewiorek92] Siewiorek, D. P., Swarz, R. S., Reliable Computer Systems Design and Implementation, Digital Press, 1992.

[Silva06] Silva, L., Madeira, H., Silva, J.G.: Software Aging and Rejuvenation in a SOAP-Based Server. In Proceedings of 5th IEEE International Simposim on Network Computing and Applications. p. 56-65. (2006)

[Skeet07] Skeet, J., Konig, D.: Groovy in Action. Manning Publications, USA (2007)

[SLAnet] Service Level Agreement and SLA Guide. (2011) [Online] http://www.service-level-agreement.net/

[SLAZone] Service Level Agreement Zone [Online], http://www.sla-zone.co.uk/index.htm (2011)

[SPECjvm98] SPECjvm98. Standard Performance Evaluation Corporation. (1998) [Online] Available: http://www.spec.org/jvm98/ (current 12 September 2011)

[Srivastava04] Srivastava, A., Eustace, A.: ATOM: a System for Building Customized Program Analysis Tools, ACM SIGPLAN Notices, v.39 n.4, 2004.

[Srivastava94] Srivastava, A.: ATOM – A System for Building Customized Program Analysis Tools, Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation, pg. 196-205, Orlando, USA, (1994)

[Sturm00] Sturm, R., Morris, W.: Foundations of Service Level Management. Sams, USA (2000).

[Subotic05] Subotić, S., Bishop, J.: Emergent Behaviour of Aspects in High Performance and Distributed Computing. Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries. p. 11-19. (2005)

[Swing] Java Swing. Oracle. [Online] Available: http://java.sun.com/javase/6/docs/technotes/guides/swing

[Tebbani06] Tebbani, B., Aib, I.: GXLA a Language for the Specification of Service Level Agreements. Lecture Notes in Computer Science, v. 4195. Springer-Verlag, Berlin Heidelberg New York. p. 201-214. (2006)

[Thwin05] Thwin, M.M.T., Quah, TS.: Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics. Journal of Systems and Software, v. 76, n. 2. Elsevier Science, New York, USA, 2005.

[Tian05] Tian, L., Noore, A.: Evolutionary Neural Network Modeling for Software Cumulative Failure Time Prediction. Reliability Engineering and System Safety, Elsevier, USA, v. 87, n. 1. p. 45-51. (2005)

[Trienekens04] Trienekens, J., Bouman, J., Zwan, M. v. d.: Specification of Service Level Agreements: Problems, Principles and Practices. Software Quality Control, v. 12 n. 1. p. 43-57. (2004)

[Urbanek11] Urbanek, S.: Low-level R to Java interface. [Online] http://www.rforge.net/rJava

[Villazon09a] Villazon, A., Binder, W., Ansaloni, D., Moret, P.: HotWave: Creating Adaptive Tools with Dynamic Aspect-Oriented Programming in Java. Proceedings of the 8th International Conference on Generative Programming and Component Engineering, Denver, USA. p. 95-98 (2009)

[Villazon09b] Villazón, A., Binder, W., Ansaloni, D., Moret, P.: Advanced Runtime Adaptation for Java. ACM SIGPLAN Notices - GPCE 09, ACM, New York, USA. p. 85-94 (2009)

[Wang04] Wang, H., Platt, J., Chen, Y., Zhang, R., Wang, Y.: PeerPressure for Automatic Troubleshooting. In Proceedings of

the Joint International Conference on Measurement and Modeling of Computer Systems. ACM, New York, New York, USA. 398-399. (2004)

[Yu05] Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In Proceedings of the ACM Symposium on Operating Systems Principles. ACM, Brighton, UK. 221-234. (2005)

[Zhuang06] X. Zhuang, M. J. Serrano, H. W. Cain, J. Choi, Accurate, efficient, and adaptive calling context profiling, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, 2006, Ottawa, Canada

Ključna dokumentacijska dokumentacija

Redni broj, **RBR**:	
Identifikacioni broj, **IBR**:	
Tip dokumentacije, **TD**:	monografska publikacija
Tip zapisa, **TZ**:	tekstualni štampani dokument
Vrsta rada, **VR**:	doktorska disertacija
Autor, **AU**:	Dušan Okanović
Mentor, **MN**:	Milan Vidaković
Naslov rada, **NR**:	Model adaptivnog sistema za praćenje i predikciju rada distribuiranih aplikacija
Jezik publikacije, **JP**:	srpski (latinica)
Jezik izvoda, **JI**:	srpski i engleski
Zemla publikovanja, **ZP**:	Srbija
Uže geografsko područje,	Vojvodina

UGP:

Godina, **GO**:	2012
Izdavač, **IZ**:	autorski reprint
Mesto i adresa, **MA**:	Novi Sad, Fakultet tehničkih nauka, Trg Dositeja Obradovića 6
Fizički opis rada, **FO**:	5/108/153/57/0/0
Naučna oblast, **NO**:	primenjene računarske nauke i informatika
Naučna disciplina, **ND**:	održavanje performansi softvera
Ključne reči, **PO**:	stalno praćenje softvera, Kieker, JMX, AOP, SLA
UDK broj, **UDK**:	
Čuva se u, **ČU**:	Biblioteka Fakulteta tehničkih nauka, Trg Dositeja Obradovića 6, Novi Sad
Važna napomena, **VN**:	nema
Izvod, **IZ**:	Stalno praćenje rada softvera je neophodno da bi se utvrdilo da li softver poštuje zadate nivoe kvaliteta. Na osnovu sakupljenih podataka, moguće je da se

predvidi i dalje ponašanje aplikacije i da se izvrši izbor daljih akcija da bi se održao zahtevani nivo. Tema ove disertacije je razvoj sistema za kontinualno praćenje performansi softvera, kao i razvoj modela za predviđanje performansi softvera. Za implementaciju sistema upotrebljena je JEE tehnologija, ali je sistem razvijen tako da može da se primeni i za praćenje softvera razvijenog za druge platforme. Sistem je modelovan tako minimalno utiče na performanse sistema softvera koji se prati. Linearna regresija je upotrebljena za modelovanje zavisnosti performansi od okruženja u kom se softver izvršava. Sistem je upotrebljen za praćenje izabrane JEE aplikacije.

Datum prihvatanja od NN veća, **DP**:	29.8.2012.
Datum odbrane, **DO**:	
Komisija, **KO**:	
predsednik:	prof. dr Dušan Surla, prof. emeritus Prirodno-matematički fakultet, Novi Sad
član:	prof. dr Zora Konjović, red. prof. Fakultet tehničkih nauka, Novi Sad
član:	doc. dr. Siniša Nešković, doc. Fakultet organizacionih nauka, Beograd

član:	prof. dr Branko Milosavljević, vanr. prof. Fakultet tehničkih nauka, Novi Sad
član:	prof. dr Milan Vidaković, vanr. prof., mentor Fakultet tehničkih nauka, Novi Sad

Keyword Documentation

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	monograph publication
Type of record, **TR**:	textual printed material
Contents code, **CC**:	doctoral dissertation
Author, **AU**:	Dušan Okanović
Menthor, **MN**:	Milan Vidaković
Title, **TI**:	Model of Adaptive System for Continuous Monitoring and Performance Prediction of Distributed Applications
Language of text, **LT**:	Serbian (latin)
Language of abstract, **LA**:	Serbian (latin)/English
Country of publication, **CP**:	Serbia

Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2012
Publisher, **PU**:	author's reprint
Publishing place, **PP**:	Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6
Physical description, **PD**:	5/108/153/57/0/0
Scientific field, **SF**:	computer sciences and informatics
Scientific discipline, **SD**:	software performance
Subject/key words, **SKW**:	continuous monitoring, Kieker, JMX, AOP, SLA
UDC:	
Holding data, **HD**:	Library of Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad
Note, **N**:	nema
Abstract, **AB**:	Continuous monitoring of software is necessary to determine whether the software performs within required service perfomance levels. Based on collected

data, it is possible to predict the future performance of applications and to plan further actions in order to maintain the required service levels. The theme of this dissertation is the development of systems for continuous performance monitoring software, as well as the development of models for predicting the performance of software. To implement the system was used JEE technologies, but the system was developed so that it can be used for tracking software developed for other platforms. The system is modeled as a minimum impact on system performance software that is monitored. Linear regression was used for modeling the dependence of the performance environment in which the software is running. The system was used to monitor selected JEE applications.

Accepted by scientific board on, **ASB**:	29.8.2012.
Defended, **DE**:	
Thesis defend board, **DB**:	
president:	prof. Dušan Surla, PhD Faculty of Science, Novi Sad
member:	prof. dr Zora Konjović, PhD Faculty of Technical Sciences, Novi Sad
member:	doc. Siniša Nešković, PhD

	Fakultet organizacionih nauka, Beograd
member:	prof. Branko Milosavljević, PhD Faculty of Technical Sciences, Novi Sad
member:	prof. Milan Vidaković, PhD, menthor Faculty of Technical Sciences, Novi Sad

Biografija

Dušan Okanović je rođen 14.03.1978. god. u Sremskoj Mitrovici, opština Sremska Mitrovica, Republika Srbija. Gimnaziju "Ivo Lola Ribar" završio je u Sremskoj Mitrovici sa odličnim uspehom.

Na odsek za Elektrotehniku i računarstvo Fakulteta tehničkih nauka u Novom Sadu upisao se 1997. godine. Studije na smeru Računarstvo i upravljanje sistemima, usmerenje Računarske nauke i informatika, završio je sa prosečnom ocenom 8,89 (osam i 89/100). Diplomski rad odbranio je 30.09.2002. godine iz predmeta "Veštačka inteligencija" sa ocenom 10 (deset). Tema diplomskog rada je bila "Evidencija logova u informacionim sistemima".

Na poslediplomske studije Fakulteta tehničkih nauka – smer Računarstvo i automatika, usmerenje Računarske nauke i informatika, upisao se školske 2002/03 godine i položio sve ispite predviđene planom i programom sa prosečnom ocenom 10 (deset). Magistarski rad pod nazivom „Sistem za upravljanje instalacijama Java aplikacija baziran na JMX tehnologiji" odbranio je 30.12.2006. godine.

U periodu od 01.10.2002 do 31.01.2004 bio je angažovan na projektima i u nastavi na Fakultetu tehničkih nauka. U zvanje asistenta-pripravnika na Fakultetu tehničkih nauka izabran je 01.02.2004. godine, a u zvanje asistenta 01.04.2007. U toku studija i rada na Fakultetu objavio je 27 naučno/stručnih radova i učestvovao na više projekta. Trenutno učestvuje u izvođenju vežbi iz predmeta: Računarski praktikum, Objektno-orijentisane tehnologije sa programiranjem i Web-dizajn. Držao je vežbe i na predmetima Osnovi računarstva i Primena računara u arhitekturi. Na Višoj tehnološkoj školi strukovnih studija u Šapcu angažovan je kao predavač na predmetima Održavanje računarskih sistema i Programiranje za internet.

Oblasti interesovanja su upravljanje softverskim aplikacijama, distribuirani sistemi i aplikacije. Aktivno se služi engleskim jezikom.

Oženjen je i živi u Novom Sadu.