УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Драган Бркин

ЈЕДНО РЕШЕЊЕ СИМУЛАЦИЈЕ ДИСТРИБУИРАНЕ СОФТВЕРСКЕ ТРАНСАКЦИОНЕ МЕМОРИЈЕ

МАСТЕР РАД

Нови Сад, 2017

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:			
Идентификациони број, **ИБР**:			
Тип документације, **ТД**:		Монографска документација	
Тип записа, **ТЗ**:		Текстуални штампани материјал	
Врста рада, **ВР**:		Дипломски – мастер рад	
Аутор, **АУ**:		**Драган Бркин**	
Ментор, **МН**:		**проф. др Мирослав Поповић**	
Наслов рада, **НР**:		**Једно решење симулације дистрибуиране софтверске трансакционе меморије**	
Језик публикације, **ЈП**:		Српски / латиница	
Језик извода, **ЈИ**:		Српски	
Земља публиковања, **ЗП**:		Република Србија	
Уже географско подручје, **УГП**:		Војводина	
Година, **ГО**:		**2017**	
Издавач, **ИЗ**:		Ауторски репринт	
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6	
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		**9/52/0/12/10/0/0**	
Научна област, **НО**:		Електротехника и рачунарство	
Научна дисциплина, **НД**:		Рачунарска техника и рачунарске комуникације	
Предметна одредница/Кqучне речи, **ПО**:		**Софтверска трансакциона меморија, Дистрибуирани системи, CloudSim**	
УДК			
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад	
Важна напомена, **ВН**:			
Извод, **ИЗ**:		**У овом раду представљено је проширење ИааС клауд симулатора КлаудСим. Прво је моделован задатак у виду трансакције над трансакционом меморијом и комуникација између центара за обраду података употребом протокола ажурирања у две фазе. Затим је имплементиран модел прототипа дистрибуиране ПСТМ. На крају је урађена евалуација и добијени резултати поређени су са претходно изведеним теоријским резултатима. Презентовани резултати су позитивни и стимулишу даљи рад у развоју дистрибуиране ПСТМ.**	
Датум прихватања теме, **ДП**:			
Датум одбране, **ДО**:		22.12.2017	
Чланови комисије, **КО**:	Председник:	**др Растислав Струхарик, ван. проф.**	
	Члан:	**др Миодраг Ђукић, доц.**	Потпис ментора
	Члан, ментор:	**др Мирослав Поповић, ред. проф.**	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Dragan Brkin**	
Mentor, **MN**:		**Miroslav Popović, PhD**	
Title, **TI**:		**A Solution for Simulation of a Distributed Software Transactional Memory**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2017	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)		**9/52/0/12/10/0/0**	
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**Software Transactional Memory, Distributed systems, CloudSim**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**This paper presents an extension of the IaaS Cloud simulator CloudSim. First task is modeled in the form of a transaction on transactional memory and communications between the data center using the Two-Phase commit protocol. Then model of a prototype of distributed PSTM is implemented. At the end evaluation has been done and obtained results are compared with the previously derived theoretical results. The presented results are positive and stimulate future work in development of distributed PSTM.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:		22.12.2017	
Defended Board, **DB**:	President:	**Rastislav Struharik, PhD**	
	Member:	**Miodrag Đukić, PhD**	Menthor's sign
	Member, Mentor:	**Miroslav Popović, PhD**	

Zahvalnost

Zahvaljujem se roditeljima na pruženoj podršci tokom studiranja i kolegi Branislavu Kordiću na strpljenju, korisnim savetima i pruženoj podršci tokom izrade ovog rada.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

CPU	*Central Processor Unit*, Centralni procesoor
TM	*Transactional Memory*, Transakciona memorija
STM	*Software Transactional Memory*, Softverska transakciona memorija
HTM	*Hardware Transactional Memory*, Hardverska transakciona memorija
PSTM	*Python Software Transactional Memory*, Pajton softverska transakciona memorija
DTM	*Distributed Transactional Memory*, Distribuirana transakciona memorija
DSTM	*Distributed Software Transactional Memory*, Distribuirana softverska transakciona memorija
DPSTM	*Distributed Python Software Transactional Memory*, Distribuirana pajton softverska transakciona memorija
VM	*Virtual Machine*, Virtualna mašina
DC	*Data Center*, Centar za obradu podataka
MI	*Million Instructions*, Milion instrukcija
MIPS	*Million Instructions Per Second*, Milion instrukcija u sekundi
RAM	*Random Access Memory*, Memorija nasumičnog pristupa
2PC	*Two-Phase Commit*, Ažuriranje u dve faze

1. Uvod

U ovom radu predstavljeno je jedno rešenje simulacije sistema distribuirane softverske transakcione memorije (engl. *Distributed Software Transactional Memory, DSTM*), kao deo projekta razvoja distribuirane pajton softverske transakcione memorije (DPSTM), koji je u toku. Generalno, softverske transakcione memorije omogućavaju mehanizme konkurentnog izvršavanja operacija, pri čemu date operacije nisu međusobno blokirajuće, jer je svaka transakcija definisana kao atomična - neprekidiva operacija. Rad se oslanja na postojeću pajton softversku transakcionu memoriju (engl. *Python Software Transactional Memory, PSTM*), razvijenu na Odseku RT-RK u Novom Sadu. Simulacija DPSTM sistema obavljena je upotrebom CloudSim simulatora, koji se koristi za simulaciju raznih procesa u distribuiranom okruženju.

Predstavljen je način funkcionisanja simulatora i ponašanje sistema. Implementiran je model prototipa DPSTM sistema i analizirani su svi nedostaci i prednosti implementacije. Poseban akcenat analize stavljen je na uticaj mreže, gde može doći do neispravnosti u radu sistema.

U drugom poglavlju detaljnije je opisana transakciona memorija, njena programska implementacija u programskom jeziku Pajton (engl. *Python*), na čiji princip rada se oslanja implementiran DPSTM sistem. Takođe, je dat opis CloudSim alata, princip rada i važnost upotrebe simulatora u procesu razvoja softvera.

U trećem poglavlju sledi analiza problema koji se javlja u distribuiranoj softverskoj transakcionoj memoriji. U četvrtom poglavlju dat je koncept rešenja modela prototipa simuliranog DPSTM sistema. U petom poglavlju izložena je programska implementacija, gde su dati detalji ključnih elementata implementacije, zajedno sa opisom najvažnijih funkcija

aplikativne sprege (engl. *Application Interface, API*). Evaluacija i način provere ispravnosti rada, odnosno verifikacija implementiranog rešenja dati su u šestom poglavlju. Rezultati testiranja prikazani su u sedmom poglavlju. Rezultati se odnose na vremena potrebna za obradu, slanje i prihvat rezultata transakcije, sa i bez modela mrežne topologije.

Zaključak ovog rada i mogući pravci daljeg unapređivanja istog dati su u poslednjem poglavlju.

2. Teorijske osnove

Kako bi objasnili implementirani sistem, njegovu funkcionalnost i značaj, potrebno je pomenuti osnove od kojih je rad započet.

2.1 Problemi i apstrakcije paralelnog programiranja

Nažalost, paralelno programiranje se dokazalo daleko komplikovanije nego programiranje sekvencijalnih programa. Algoritmi paralelnih programa su teži za postavku i dokazivanje ispravnosti od ekvivalentnih sekvencijalnih algoritama. Paralelni program je mnogo teži za dizajn, pisanje i kontrolisano izvršavanje nego ekvivalentni sekvencijalni program. Nedeterminističke greške koje se javljaju u paralelnim programima je ozbiljno teško pronaći i popraviti. Na kraju paralelni programi su često loše izvedeni. Međutim, ovo objašnjenje leži u delu problema da ljudi imaju mnogo poteškoća sa praćenjem više događaja koji se događaju istovremeno. Paralelizam i neodređenost znatno povećavaju broj stavki koje programer softvera mora imati na umu. Zbog toga, malo ljudi je u stanju da sistematski razmišlja o ponašanju paralelnog programa. Primarna motivacija za ozbiljno interesovanje za transakcione memorije - programski modeli, jezici i alati dostupni paralelnom programeru zaostajali su daleko iza onih za sekvencijalne programe. Postoje dva preovladavajuća paralelna programska modela: paralelizam podataka i paralelizam zadataka. Paralelizam podataka je efikasan programski model koji istovremeno primenjuje operaciju na zbir pojedinačnih stavki [1]. Posebno je pogodan za numeričke račune, koje koriste matrice kao njihove primarne strukture podataka. Programi često manipulišu matricom kao agregat, na primer, dodajući ga drugoj matrici.

Naučni jezici za programiranje, kao što je High Performance Fortran (HPF) [2], direktno podržavaju paralelno programiranje podataka sa skupom operatora na matricama i načinima za

kombinovanje ovih operacija. Paralelizam je implicitan i obilan u paralelnim programima podataka. Kompajler može iskoristiti inherentnu konkurentnost primene operacije na elemente agregata deljenjem rada između raspoloživih procesora. Ovaj pristup pomera teret sinhronizacije i balansiranja opterecenja od programera do sistema za prevođenje i izvršavanje. Nažalost, paralelizam podataka nije univerzalni programski model. To je prirodno i praktično u nekim postavkama [1], ali se teško primenjuje na većinu struktura podataka i programskih problema Drugi zajednički programski model je paralelizam zadataka koji izvršava zadatke na odvojenim nitima koji su usklađeni sa eksplicitnom sinhronizacijom kao što su fork-join operacije, brave (engl. *Locks*), semafori (engl. *Semaphores*), redovi (engl. *Queues*) itd. Ovaj model nestruktuiranog programiranja ne nameće ograničenja na kod koji svaka nit izvršava, kada ili kako nit komunicira, ili kako se zadaci dodeljuju nitima. Model je opšti, sposoban da izrazi sve oblike paralelnog izračunavanja. Međutim, vrlo je teško pravilno programirati. Na mnogo načina, model je na istom (malom) nivou apstrakcije kao hardver računara. Procesori direktno implementiraju mnoge konstrukcije koje se koriste za pisanje ove vrste programa [3].

2.1.1 Apstrakcija paralelnog programiranja

Ključni nedostatak paralelizma zadataka je nedostatak efikasnih mehanizama za apstrakciju i dva osnovna alata za kompjutersku nauku za upravljanje složenošću. Apstrakcija je pojednostavljen pogled na entitet, koji obuhvata karakteristike koje su neophodne za razumevanje i manipulaciju za određenu svrhu. Ljudi sve vreme koriste apstrakciju. Apstrakcija skriva nebitne detalje i složenost i omogućava ljudima (i računarima) da se fokusiraju na aspekte problema koji su relevantni za određeni zadatak.

Sastav je sposobnost sastavljanja dva entiteta kako bi se formirao veći, složeniji entitet, koji je, pak, apstrahovan u jedan, kompozitni entitet. Sastav i apstrakcija su blisko povezani, jer se detalji o osnovnim entitetima mogu suprimisati prilikom manipulisanja sa kompozitnim proizvodom.

Savremeni programski jezici podržavaju snažne mehanizme apstrakcije, kao i bogate biblioteke apstrakcija za sekvencijalno programiranje. Postupci nude način kako da enkapsuliraju i imenuju niz operacija. Apstraktni tipovi podataka i objekti nude način da enkapsuliraju i imenuju strukture podataka. Biblioteke, okviri (engl. *Framework*) i modeli dizajna prikupljaju i organizuju višekratne apstrakcije koje su građevinski blokovi softvera. Povećavanje nivoa apstrakcije, kompleksni softverski sistemi, kao što su operativni sistemi, baze podataka ili međuslovni softver, pružaju moćne, generalno korisne apstrakcije, kao što su virtuelna memorija, datotečni sistemi ili relacijske baze podataka koje koristi većina softvera. Ovi mehanizmi za

apstrakciju i apstrakcije su fundamentalni za savremeni razvoj softvera koji sve više gradi i ponovo koristi programske komponente, umesto da ih piše iz nule.

Paralelnim programiranjem nedostaju uporedivi mehanizmi apstrakcije. Paralelni programski modeli niskog nivoa, kao što su niti i eksplicitna sinhronizacija, nisu prikladni za konstrukciju apstrakcija jer eksplicitna sinhronizacija nije kompozabilna. Program koji koristi apstrakciju koja sadrži eksplicitnu sinhronizaciju mora biti svestan njegovih detalja, kako bi se izbeglo izazivanje trka (engl. *Races*) ili blokada (engl. *Deadlocks*) [3].

2.2 Sistemi baza podataka i transakcije

Dok je paralelizam bio težak problem za programiranje opštih namena, sistemi baze podataka uspešno koriste paralelni hardver decenijama. Baze podataka postižu dobre performanse izvršavajući mnogo upita istovremeno i izvršavajući upite na više procesora kada je to moguće. Štaviše, model programiranja baze podataka osigurava da se autor pojedinačnog upita ne mora brinuti o tom paralelizmu. U srcu programskog modela za baze podataka je transakcija. Transakcija određuje semantiku programa u kojoj se računanje izvršava kao da je to jedino računanje koje pristupa bazi podataka. Ostali računi mogu se izvršiti istovremeno, ali model ograničava dozvoljene interakcije između transakcija, tako da svaki proizvodi rezultate koji se ne mogu razlikovati od situacije u kojoj se transakcije pokrivaju jedno za drugim. Ovaj model je poznat kao serializabilnost. Kao posledica toga, programer, koji piše kod za transakciju živi u jednostavnom, poznatom sekvencijalnom programskom svetu i mora samo da razume račune koji počinju sa konačnim rezultatima drugih transakcija. Transakcije dozvoljavaju istovremene operacije da pristupe zajedničkoj bazi podataka i još uvek proizvode predvidljive, ponovljive rezultate. Transakcije se implementiraju pomoću osnovnog sistema baze podataka ili monitora za obradu transakcija, od kojih obe sakrivaju kompleksne implementacije iza relativno jednostavnog interfejsa [4][5]. Ovi sistemi sadrže mnoge sofisticirane algoritme, ali programer samo vidi jednostavan programski model koji podrazumeva većinu aspekata istovremenosti i neuspeha. Štaviše, apstraktna specifikacija ponašanja transakcije obezbeđuje veliku slobodu implementacije i omogućava izgradnju efikasnih sistema baza podataka. Transakcije nude dokazan mehanizam apstrakcije u sistemima baza podataka za konstrukciju višekratnih paralelnih računanja. Računanje izvršeno u transakciji ne treba izlagati podacima kojima pristupa ili redosledu u kojem se ovi pristupi javljaju. Sastavljanje transakcija može biti jednako jednostavno kao izvršavanje podtransakcije u okviru okolne transakcije. Mehanizmi koordinacije pružaju koncizne načine da ograniče i narede izvršenje istovremenih transakcija. Pojava višejezgarnih procesora je obnavljala interesovanje za staru ideju, uključivanje transakcija u programski model koji se koristi za pisanje paralelnih programa - izgradnja ideja iz jezika kao

što je Argus, koji su obezbedili transakcije kako bi pomogli strukturiranim distribuiranim algoritmima [6]. Dok transakcije na programskom jeziku nose sličnost sa ovim transakcijama, okruženja za implementaciju i izvršenje se značajno razlikuju, pošto operacije u distribuiranim sistemima obično uključuju mrežnu komunikaciju, a operacije u transakcione baze podataka obično uključuju pristup disku. Nasuprot tome, programi obično čuvaju podatke u memoriji. Ova razlika je dala ovoj novoj apstrakciji svoje ime, transakciona memorija (engl. *Transaction Memory, TM*).

2.2.1 Transakcija

Transakcija je sekvenca akcija koje su nevidljive nekom spoljašnjem posmatraču. Transakcija baze podataka ima četiri specifična atributa: atomsko ponašanje, konzistentnost, izolaciju i izdržljivost - kolektivno poznate kao ACID svojstva.

Atomičnost zahteva da se sve konstituisane akcije u transakciji uspešno završe ili da se ni jedna od ovih akcija ne počine izvršavati. Nije prihvatljivo da konstitutivna akcija ne uspe i da se transakcija uspešno završi. Niti je prihvatljivo da neka akcija ostane iza dokaza da je izvršena. Transakcija koja se uspešno završi komituje (engl. *Commits*), a ona koja ne uspe prekine se (engl. *Aborts*).

Sledeća osobina transakcije je konzistentnost. Značenje konzistentnosti je potpuno zavisno od aplikacije, a obično se sastoji od skupa invarijantnih podataka na strukturama podataka.

Ako transakcija modifikuje stanje objekta, onda njegove promene trebaju započeti iz jednog konzistentnog stanja i ostaviti bazu podataka u drugom konzistentnom stanju. Kasnije transakcije možda nemaju znanje o tome koje transakcije su izvršene ranije, pa je nerealno očekivati da ih pravilno izvršavaju ako invarienti koje očekuju nisu zadovoljni. Održavanje konzistentnosti je trivijalno zadovoljeno ako se transakcija prekine, jer tada ne uznemirava konzistentno stanje u kojem je počelo.

Sledeća osobina, nazvana izolacija, zahteva da transakcije ne ometaju jedna drugu dok rade - bez obzira na to da li se paralelno izvršavaju ili ne. Ova osobina očigledno čini transakcije atraktivnim programskim modelom za paralelne računare.

Izdržljivost zahteva da nakon što se transakcija izvrši, njen rezultat bude trajan i dostupan svim naknadnim transakcijama [3].

2.3 Transakciona memorija

Transakciona memorija (engl. *Transactional Memory, TM*) predstavlja paradigmu, odnosno mehanizam paralelnog programiranja za više jezgarne i nadolazeće mnogojezgarne arhitekture. U računarstvu transakione memorije imaju zadatak da pojednostave konkurentno

programiranje omogućavajući izvršavanje grupe instrukcija čitanja i pisanja kao jednu nedeljivu – atomsku instrukciju.

1977.godine, Lomet je zapazio da apstrakcija slična transakciji sa bazom podataka može napraviti dobar programski jezički mehanizam kako bi se osigurala konzistentnost podataka podeljenih između nekoliko procesa [7]. Rad nije opisao praktičnu implementaiju koja je konkurentna sa eksplicitnom sinhronizacijom, tako da je ideja postojala do 1993. kada su Herlihi i Moss [8] predložili hardverski podržanu transakcionu memoriju, a Stone i drugi [9] predložili operaciju sa atomskim operacijama višem rečima poznatu pod nazivom „Oklahoma Update. Poslednjih godina postojalo je veliko intresovanje za hardverske i softverske sisteme za implementaciju transakcione memorije.

Osnovna ideja je vrlo jednostavna. Karakteristike transakcija pružaju pogodnu apstrakciju za koordinaciju istovremenih čitanja i pisanja podataka o istovremenim ili paralelnim sistemima. Danas je ova koordinacija odgovornost programera koji ima samo mehanizme nižeg nivoa, kao što su brave, semafori, muteksi itd, kako bi se sprečilo da se dve uzastopne niti mešaju. Čak i savremeni jezici kao što su Java i C# obezbeđuju samo nešto viši nivo konstrukcije da bi se sprečio istovremeni pristup unutrašnjim podacima objekta. Kao što je ranije rečeno, ovi mehanizmi niskog nivoa su teško upotrebljivi i nisu kompozabilni.

Transakcije pružaju alternativni pristup koordinaciji istovremenih niti. Program može obaviti računanje u transakciji. Atomično ponašanje osigurava da se računanje uspešno završi i izvrši svoj rezultat u potpunosti ili prekida. Pored toga, izolacija osigurava da transakcija proizvodi isti rezultat kao i ako se ne izvršavaju druge transakcije istovremeno. Iako se čini da je izolacija osnovna garancija za transakcionu memoriju, i druga svojstva, atomska svojstva i konzistentnost, su važni. Ako je cilj programera ispravan program, onda je konzistentnost važna jer se transakcije mogu izvršiti u nepredvidivom redosledu. Bilo bi teško napisati ispravan kod bez pretpostavke da transakcija počinje da se izvršava u doslednom stanju. Neuspeh atomskog izvršavanja je ključni deo osiguranja doslednosti. Ako transakcija ne uspe, mogla bi ostaviti podatke u nepredvidljivom i neusaglašenom stanju koje bi uzrokovale neuspeh naknadnih transakcija. Mehanizam koji se koristi za implementaciju neuspeha atomskog izvršavanja vraća podatke u ranije stanje, što je veoma važno za implementaciju određenih tipova kontrole konkurencije [3].

2.3.1 Softverska Transakciona Memorija

Softverska Transakciona Memorija (engl. *Software Transactional Memory*) – STM predstavlja softversku implementaciju transakcione paradigme paralelnog programiranja. STM sistemi pružaju visok nivo apstrakcije nasuprot niskom nivou sinhronizaciji niti. Ovakva

apstrakcija omogućava lakšu koordinaciju između istovremenog čitanja i pisanja nad podacima u paralelnim sistemima [3].

Rečnici, koji se nalaze u privatnom skladištu TM, obezbeđuju STM sistem svim informacijama potrebnim za otkrivanje konflikata i njihovo rešavanje. Broj verzije u istoriji čitanja omogućava niti da otkrije da li je istovremena nit ažurirala objekat od koga je pročitao, zbog toga što će ažuriranje povećati broj verzije objekta. Vrednosti snimljene u rečniku dozvoljavaju niti da poništi transakciju ako se pojavi ova vrsta sukoba.

Razlike u osnovnom računarskom hardveru mogu znatno uticati na performanse sistema STM-a, na primer, model konzistentnosti memorije koji hardver pruža i troškove sinhronizaionih operacija u poređenju sa običnim pristupom memoriji.

U kojoj meri su STM sistemi dovoljno brzi za korišćenje u praksi ostaje sporno istraživačko pitanje. U opisanom jednostavnom sistemu, STM sistem uvodi dodatne radove za održavanje rečnika i obavljanje sinhronizacijskih operacija na objektima.

2.3.2 Hardverska transakciona memorija

Bez obzira da li je STM dovoljno brza sama po sebi, jasno je da postoji mogućnost značajnog poboljšanja performansi pomocu hardverske podrške.

Rani sistemi hardverske transakcione memorije (engl. *Hardware Transactional Memory, HTM*) su držali modifikovano stanje transakcije u keš memoriji (engl. *Cache Memory*) i koristili protokol za koherentnost keša za otkrivanje sukoba sa drugim transakcijama. Nedavni HTM sistemi su istraživani pomoću bafera za pisanje procesora za održavanje transakcija ili prenošenjem transakcijskih podataka u niže nivoe hijerarhije memorije ili u memoriju koja se upravlja softverom.

HTM sistemi obično pružaju primitivne mehanizme koji podležu korisničkim jezicima, kompajlerima i sistemima izvršavanja. Softver premošćava jaz između programera i hardvera, što čini veliki deo diskusije o STM sistemima, jezicima i kompajlerima koji su relevantni i za HTM sisteme [3].

2.3.3 Pajton Softverska Transakciona Memorija

Pajton softverska transakciona memorija – PSTM (engl. *Python Software Transactional Memory*) je realizovana pomoću rečnika unutar servera, odnosno centra za obradu podataka. Svakom korisniku se, na zahtev, prosleđuje kopija transakcione promenljive (t-promenljive) i na zahtev, ukoliko je to moguće, ažurira se stanje rečnika. Za prijem, odnosno slanje zahteva koristi se red (engl. *Queue*) kako bi se zahtevi mogli obrađivati po redosledu kojim su poslati [10]. Klijenti koji koriste transakcionu memoriju zovu se transakcione aplikacije.

PSTM-zasnovana aplikacija sadrži skup transakcija koje obavljaju operacije nad lokalnim promenjivama, odnosno kopijama t-promenljivih. Transakcija zahteva kopije t-promenljivih na početku i ažurira neke od njih na kraju, koristeći PSTM API spregu. Ovu spregu čini skup javnih funkcija definisanih u PSTM modulu koje zahtev za obradu šalju udaljenom serveru upotrebom RPC (engl. *Remote Procedure Call*) mehanizma. Rečnik, koji se koristi je skup torki (key, ver, val), gde je key naziv t-promenljive, ver trenutna verzija, a val vrednost. PSTM server poslužuje dolazne zahteve automatski. Po prijemu, zahtev se obradi i odgovor se odmah šalje nazad klijentu [11].

2.4 Računarstvo u oblaku

Računarstvo u oblaku (engl. *Cloud Computing*) je tehnologija koja podržava novi način korišćenja hardverske infrastrukture. Dobavljači oblaka nude ove infrastrukture u vidu virtuelnog hardvera upravljanog od strane odgovarajućeg softvera. Oni nude svoje usluge u formi virtuelnih mašina (VM) spremnih za korišćenje na zahtev korisnika [12] i kao takve predstavljaju osnovu za distribuirane sisteme.

Infrastruktura kao usluga – IaaS (engl. *Infrastructure as a Service*) predstavlja jedan od tri osnovna modela u računarstvu u oblaku. Ovaj model omogućava korišćenje računarske infrastrukture u vidu VM. Korisnik je u mogućnosti da upravlja VM, njihovim umrežavanjem kao i skladištenjem podataka. U virtuelnoj infrastrukturi korisnik može pokrenuti različite vrste programske podrške, od operativnog sistema do aplikacija. Za pristup infrastrukturi koristi se internet. Da bi ovaj servis bio dostupan korisnicima, neophodan je softver koji omogućava administraciju infrastrukture, jednostavno dodeljivanje resursa, upravljanje infrastrukturom i merenje performansi.

2.5 Distribuirani naspram centralizovanih sistema

Kod centralizovanih sistema svi podaci se nalaze na jednom (centralnom) računaru, ali im svi korisnici mogu pristupati, sa bilo kog računara, ukoliko su mrežno povezani.

Kod distribuiranih sistma, s druge strane, podaci su razbacani na više računara, koji mogu biti geografski prilično udaljeni jedan od drugog.

Korisnik nema informaciju da li se podaci koje traži nalaze na jednom centralnom ili su podeljeni na nekoliko distribuiranih računara. S toga, sa stanovišta korisnika razlika između centralizovanih i distribuiranih sistema nije vidljiva.

Jedna od prednosti distribuiranog sistema je znatno veći kapacitet koji se ogleda kako u memoriji tako i u procesorskoj moći. Takođe, smanjen je protok informacija kroz

komunikacione kanale računarske mreže, jer ne pristupa svaki korisnik svakoj bazi podataka u sistemu, čime se smanjuje mogućnost zagušenja u mreži.

Prava eksplozija centara za obradu podataka došla je tokom 1997-2000 godina. Kompanijama je bilo potrebno brzo internet povezivanje i konstantno uspostavljanje prisustva na internetu. Instaliranje takve opreme nije bilo održivo za mnoge manje kompanije. Mnoge kompanije započele su izgradnju veoma velikih objekata, pod nazivom Internet centri za obradu podataka (engl. *Internet Data Centers, IDCs*), koji komercijalnim klijentima pružaju niz rešenja za primenu i rad sistema. Nove tehnologije i prakse su dizajnirane da se bave skalom i operativnim zahtevima takvih velikih operacija. Ove prakse su na kraju migrirale prema privatnim centrima podataka, i usvojene su uglavnom zbog njihovih praktičnih rezultata. Centri za obradu podataka za računarstvo u oblaku se nazivaju centri za obradu podataka u oblaku (engl. *Cloud Data Centers, CDCs*). Ali danas, podela ovih uslova gotovo je nestala i integrišu se u termin "data centar"[13][14].

Sa povećanjem korišćenja računarstva u oblaku, poslovne i vladine organizacije detaljnije istražuju centre podataka u oblastima kao što su sigurnost, dostupnost, uticaj na životnu sredinu i poštovanje standarda. Dokumenti o standardima akreditovanih profesionalnih grupa, kao što je Asocijacija telekomunikacijske industrije, određuju zahteve za dizajn centara podataka. Poznati operativni metriki za dostupnost podataka centra mogu služiti za procenu komercijalnog uticaja prekida. Razvoj se nastavlja u operativnoj praksi, kao i u dizajnu centra za ekološki prihvatljive podatke.

Centralizovani centri za obradu podataka obično koštaju mnogo za izgradnju i održavanje.

2.6 CloudSim Alat

CloudSim predstavlja jedan od alata za modelovanje i simulaciju okruženja računarskog oblaka i evaluaciju algoritama za obezbeđivanje resursa [12][15].

Korišćen je za simulaciju i analizu aplikacija velikih razmera poput društvenih mreža u oblaku [15], evaluaciju strategija za raspoređivanje poslova zasnovanih na SLA (engl. *Service Level Agreement*) u okviru distribuiranih i centara za obradu podataka u oblaku [16], kao i za analizu performansi i energetsku procenu I/O (engl. *Input/Output*) operacija unutar centara za obradu podataka na osnovu VM [12].

Ovaj alat radi kao simulator zasnovan na diskretnim događajima i implementiran je u programskom jeziku Java. Osnovni elementi CloudSim-a čine entiteti: CloudInformationService, Datacenter, DatacenterBroker i CloudsimShutdown. Ovi entiteti predstavljaju osnove elemente arhitekture centra za obradu podataka (engl. *Datacenter*). Komunikacija između entiteta se ostvaruje slanjem definisanih poruka događaja (npr. VM_CREATE, CLOUDLET_SUBMIT itd). Ovi događaji mogu biti spoljašnji (poslati od strane jednog entiteta drugom) ili unutrašnji (poslati i primljeni od strane istog entiteta). Po primanju, svaka od poruka događaja se preuzima i sprovode se određene akcije, pre nego što se pošalje poruka potvrde (npr VM_CREATE_ACK) [12]. Simulacija u CloudSim-u zasnovana je na izvršavanju objekta klase Cloudlet. Cloudlet modeluje proces opisan veličinom ulaznih i izlaznih podataka, potrebnom RAM memorijom, brojem procesorskih jezgara kao i opterećenja procesora, načinom raspoređivanja, itd. Cloudlet se izvršava na VM u okviru centra za obradu podataka. Broker modeluje krajnjeg korisnika koji serveru, odnosno VM prosleđuje proces i prihvata rezultat obrade.

2.6.1 Upotreba simulatora

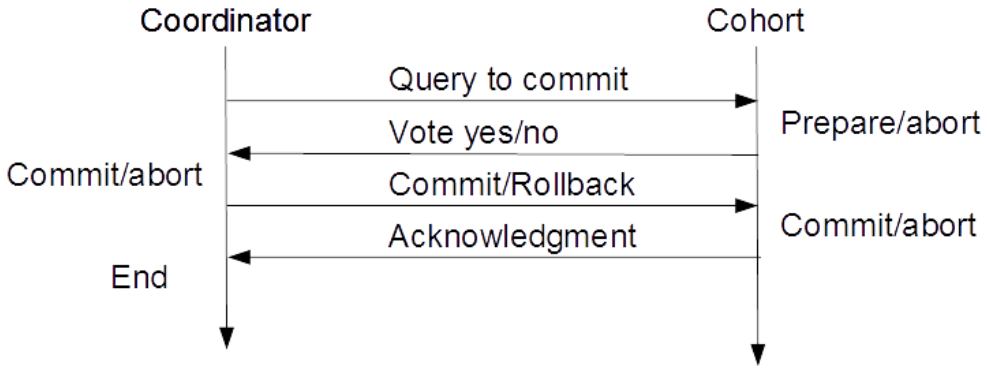
Zbog čega koristiti simulator umesto pravog uređaja?

U avio industriji simulatori se koriste za obuku pilota, treniranje specijalnih manevra, što ima dvostranu korist. Prva je ušteda resursa, a druga je izbegavanje mogućih nezgoda. Slično ovome, upotreba raznih tipova simulatora u računarstvu nije strana. Od simulacije raznih računarskih arhitektura, operativnih sistema, pa sve do složenih sistema sačinjenih od više računarskih uređaja. Ovime se postiže ušteda resursa, ili pak izbegavanje mogućih neželjenih ishoda i nepravilnosti u projektovanju sistema, što može dovesti do finansijskih katastrofa.

Upravo iz svega prethodno navedeno, upotreba simulatora je veoma važan i ozbiljan zadatak u procesu projektovanja i implementacije novih računarskih protokola, arhitektura, operativnih sistema, mrežnih arhitektura i uređaja.

2.7 Protokol ažuriranja u dve faze

Na slici 2.1 predstavljena je jednostavna komunikacija protokola ažuriranja u dve faze (engl. *Two-Phase Commit Protocol, 2PC protocol*).



Slika 2.1Protokol ažuriranja u dve faze

Server 1 (Coordinator) šalje zahtev za pripremu serveru 2 (Cohort). Nakon čega server 2 odgovara pozitivno ili negativno. U zavisnosti od odgovora koji primi, server 1 će poslati zahtev za ažuriranje ili prekid. Po izvršenoj akciji, server 2 šalje potvrdu o izvršenoj akciji.

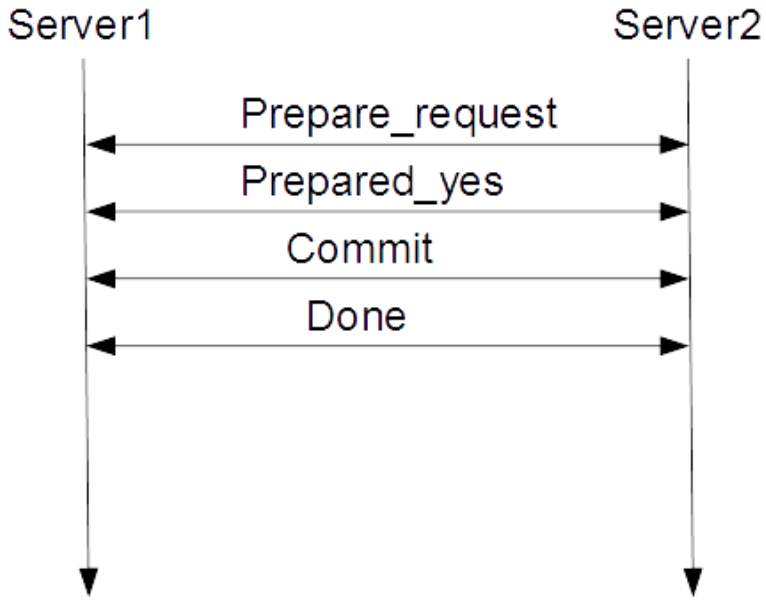
3. Analiza problema

Glavni problem simulacije distribuirane pajton softverske transakcione memorije (DPSTM) leži u tome kako u CloudSim alatu predstaviti transakcioni sistem koji čine: transakcije i operacije koje one mogu da obavlju, transakcione promenljive i komunikaciju između centara za obradu podataka i korisnika. Kako je već prethodno pomenuto, transakcija nad skupom transakcionih promenljivih (t-promenljivih) može da obavlja operacije čitanja i/ili pisanja, pa je s toga prvo potrebno osmisliti model transakcione promenljive, koji će činiti osnovni gradivni deo svake transakcije.

Kao sledeća prepreka javlja se model korsničke sprege pomoću koje će korisnici udaljenom serveru poslati zahtev za obradu. Na kraju, ali ne i najmanje važno, postavlja se pitanje kako modelovati komunikaciju između udaljenih servera, odnosno centara za obradu podataka. Kao jednostavno, ali praktično rešenje za komunikaciju između servera uzima se protokol ažuriranja u dve faze (engl. *Two Phase Commit Protocol, 2PC Protocol*). Ovaj protokol se pokreće svaki put kada korisnik pošalje serveru zahtev za ažuriranje, jer je tada potrebno dovesti podatke na svakom serveru u konzistentno stanje, odnosno usaglasiti kopije transakcionih promenljivih na svim serverima u sistemu. Međutim, pojednostavljena implementacija ovog protokola, bez koordinatora, može dovesti sistem u nedefinisano stanje, pa čak i dovesti do blokiranja u sistemu (engl. *Livelock*).

Na slici 3.1, pomoću MSC dijagrama, prikazan je jedan primer nedefinisanog stanja uzrokovanog upotrebom pojednostavljenog 2PC protokola. Neka u sistemu postoje dva servera istih performasni i neka je kašnjenje poruke sa obe strane jednako. Ukoliko se na oba servera istovremeno pojavi zahtev za ažuriranje istog skupa t-promenljivih dešava se sledeće. Oba servera lokalno obrađuju zahtev. Kako su serveri istih performansi, lokalna obrada zahteva će se

završiti za istu količinu vremena i 2PC protokol će se pokrenuti istovremeno i na serveru 1 i na serveru 2. Poruke koje serveri šalju jedan drugom će, zbog jednakog mrežnog kašnjenja, biti istovremeno poslate i istovremeno primljene. Nakon prijema sledi obrada zahteva i slanje potvrdnog odgovora. Isti skup t-promenljivih je ažuriran na oba servera, ali podaci nisu konzistentni.



Slika 3.1Problem jednostavnog rešenja 2PC protokola

Problem je sledeći: oba servera će uspešno obaviti ažuriranje istog skupa t-promenljivih, iako ažuriranje ne mora biti isto. Npr. server 1 ažurira skup t-promenljivih na sledeći način: x+y = z; y/x = w, dok server 2 isti skup t-promenljivu ažurira na sledeći način x+y = w, y/x = z. Oba korisnika će dobiti informaciju o uspešnoj transakciji, ali će na serverima biti različito stanje istih t-promenljivih:

TABELA 3.1

PRIMER NEKONZISTENTNOSTI PODATAKA NA SERERIMA NAKON AŽURIRANJA

Pre ažuriranja		Nakon ažuriranja	
DC1	DC2	DC1	DC2
(X,0,1)	(X,0,1)	(X,0,1)	(X,0,1)
(Y,2,5)	(Y,2,5)	(Y,2,5)	(Y,2,5)
(Z,4,1)	(Z,4,1)	(Z,5,6)	(Z,5,5)
(W,8,93)	(W,8,93)	(W,9,5)	(W,9,6)

3.1 Koordinator

Prethodno prikazan primer ilustruje dobar temelj za pojavu blokiranja sistema (engl. *Livelock*). Slično kao i pri pojavi međusobnog blokiranja (engl. *Deadlock*), dolazi do neželjenog rada sistema. Procesi uključeni u blokiranju sistema su u mogućnosti da konstantno menjaju svoje stanje, jedan u odnosu na drugi, ali ni jedan od njih ne napreduje.

Kao jedno od rešenja pomenutog problema je postavljanje koordinatora u sistem. Njegova uloga bi bila rukovanje komunikacijom između servera i njihovim usaglašavanjem. Princip rada i detaljniji opis koordinatora biće objašnjen u narednom poglavlju.

3.1.1 Izbor lidera

U distribuiranom računarstvu, izbor lidera (koordinatora) je proces označavanje jednog čvora kao koordinatora nekog zadatka koji se distribuira između nekoliko čvorova. Pre nego što se zadatak započne, nisu svi mrežni čvorovi ili svesni koji je od njih koordinator zadatka ili ne mogu da komuniciraju sa trenutnim koordinatorom. Nakon pokretanja algoritma za izbor lidera, svaki čvor u mreži prepoznaje određeni, jedinstveni čvor kao koordinatora zadatka.

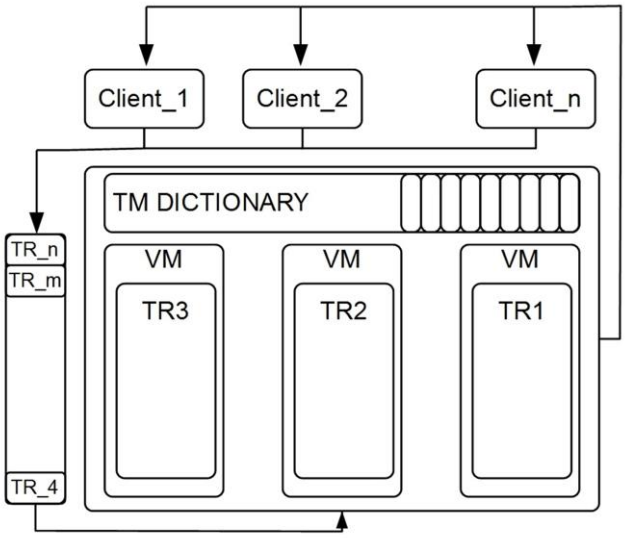
U nastavku su pobrojani neki od algoritama koji se koriste za izbor lidera:

1. Izbor lidera u prstenu:
    a. Anonimni prsten,
        i. Sinhroni algoritam
        ii. Asinhroni algoritam
    b. Eponimni prsten,
        i. Sinhroni algoritam
        ii. Asinhroni algoritam
2. Uniformni algoritam,
3. Neuniformni algoritam
4. Algoritam randomiziranog izbora lidera

Izbor lidera u projektovanom DPSTM sistemu nije implementiran. Pretpostavlja se da je lider unapred izabran i poznat.

4. Koncept rešenja

Opisani problemi u prethodnom poglavlju rešeni su proširivanjem postojećih paketa dostupnih u CloudSim alatu, neophodnih za simulaciju distriburane pajton softverske



Slika 4.1Blok dijagram servera u DSTM

transakcione memorije (DPSTM).

Na slici 4.1 prikazan je blok dijagram jednog servera u simuliranom DPSTM sistemu. Svaki server sadrži kopije transakcionih promenljivih koje su smeštene u rečniku (engl. *Transactional Memory Dictionary, TM Dictionary*). Sve nadolazeće transakcije, upućene istom serveru, smeštaju se u ulazni red (engl. *Input Queue*) tog servera, odakle bivaju raspoređene u virtualne mašine (engl. *Virtual Machine, VM*). Po završenoj obradi, rezultat transakcije vraća se klijentu koji je zahtev poslao.

Broj transakcija (brTR) koje se mogu paralelno obrađivati na jednom serveru iznosi: *brTR = TRpVM * VMpS*, gde je *TRpVM* broj transakcija koje se mogu paralelno obrađivati na istoj VM, a *VMpS* broj VM koje mogu biti pokrenute na jednom serveru.

4.1 Transakciona promenljiva

Transakciona promenljiva, t-promenljiva, predstavlja osnovni gradivni element transakcije. Nad t-promenljivom moguće je obavljati operacije čitanja i pisanja. Pod pisanjem nad t-promenljivom podrazumeva se ažuriranje trenutnog stanja iste, dok čitanje predstavlja regularno čitanje trenutnog stanja t-promenljive.

Transakciona promenljiva predstavljena je kao uređen par: (*ključ*, *verzija*), gde je *ključ* jednistveni identifikator, odnosno naziv transakcione promenljive, dok je *verzija* trenutna verzija transakcione promenljive. Vrednost t-promenljive je izostavljna, jer nije relevantna za potrebe simulacije. Model transakcione promenljive implementiran je u klasi *Tvar.java*. Detalji implementacije dati su u sledećem poglavlju.

4.2 Model transakcije

Transakcija je modelovana kao nedeljiva operacija na visokom nivou apstrakcije. Ovime se od korisnika sakrivaju operacije i uslovi neophodni za izvršavanje transakcije, bilo ona uspešna ili neuspešna. Transakcija sadrži dva skupa t-promenljivih. Jedan nad kojem se radi validacija, a drugi koji se ažurira. Ovi skupovi mogu biti isti, a mogu biti i potpuno različiti.

Ukoliko su svi uslovi zadovoljeni, transakcija će se uspešno izvršiti i korisnik će dobiti izveštaj o uspešno završenoj transakciji, u suprotnom korisniku se šalje izveštaj o neuspešno izvršenoj transakciji i prepušta mu se sloboda daljeg odlučivanja.

Nakon dobijenog odgovora, korisnik je u mogućnosti da zatraži najnovije verzije t-promenljivih nad kojima želi da operira, a kasnije da obavi validaciju ili ažuriranje istih.

4.3 Model servera (centar za obradu podataka)

Server DPSTM sistema predstavljen je kao centar za obradu podataka, na kojem su pokrenute VM. Unutar ovih VM obrađuju se transakcije, odnosno zahtevi koje korisnici upotrebom API sprege šalju serveru.

Svaki zahtev se obrađuje na zasebnoj VM, da bi se obezbedile najbolje performanse koje server može da pruži. Drugi razlog pokretanja zasebne VM za svaki primljeni zahtev je raspoređivanje zadataka unutar VM. Kako je transakcija predstavljena kao jedinstvena nedeljiva

operacija, nije moguće dve ili više transakcija obrađivati na jednom procesoru međusobnim učešljavanjem, niti se prebacivati sa jednog na drugi proces u toku obrade transakcije.

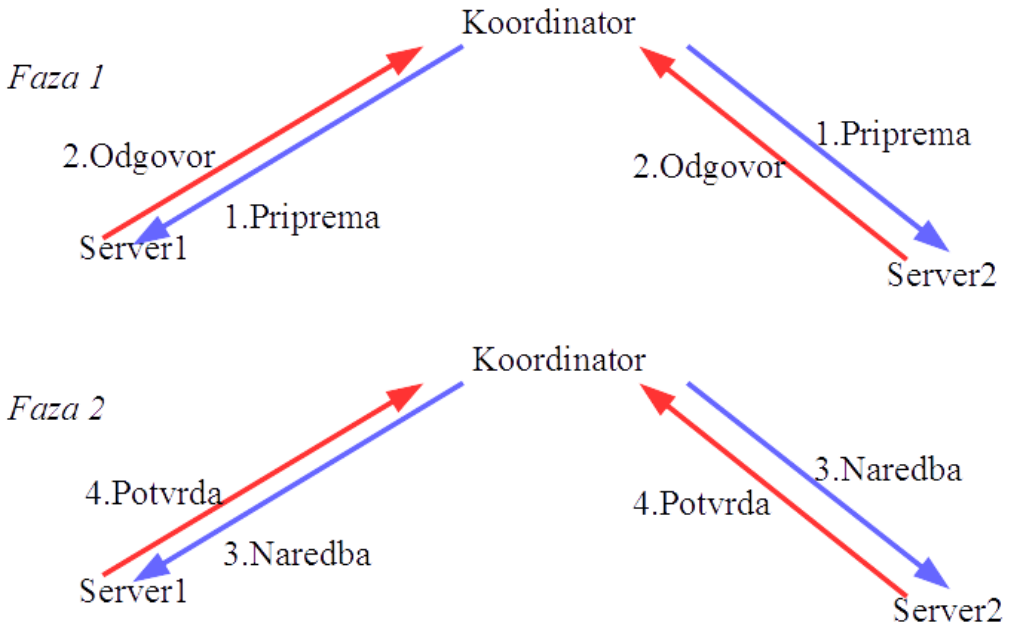
4.3.1 Model transakcione memorije

Svaki server u DPSTM sistemu poseduje lokalnu kopiju transakcione memorije. Ključna stavka je usaglašenost svih kopija transakcione memorije na svakom serveru u sistemu. Ovim se postiže osećaj jedinstvenosti i prividno stanje da u sistemu postoji samo jedan server na kojem se nalazi baza podataka nad kojom klijenti, odnosno korisnici putem transakcija operiraju.

Kopija transakcione memorije na svakom serveru modelovana je kao mapa, gde je ključ naziv t-promenljive, a vrednost ključa je trenutna verzija. Kako je stvarna vrednost t-promenljive, sa kojom bi korisnici obavljali interne operacije, za potrebe simulacije irelevantna, ona se izostavlja i ne uzima se u razmatranje.

4.3.2 Koordinator

U poglavlju Analiza problema, pomenut je koordinator DPSTM sistema. Uloga koordinatora je preuzimanje komunikacije među serverima u trenutku ažuriranja neke od t-promenljive u transakcionoj memoriji. Na slici 4.2 ilustrovana je komunikacija koordinatora sa dva servera.



Slika 4.2Komunikacija koordinatora i servera

U trenutku kada na bilo koji server pristigne zahtev za ažuriranje jedne ili skupa t-promenljivih dešava se sledeće:

• Server koji je primio zahtev, proverava svoju lokalnu kopiju transakcione memorije kako bi utvrdio da li korisnik obavlja operacije nad validnim t-promenljivama.

• Nakon lokalne provere, server se obraća koordinatoru slanjem zahteva za pokretanje 2PC protokola.

• Po prijemu zahteva za pokretanje 2PC protokola, koordinator svim serverima u sistemu, uključujući i server od kojeg je zahtev primio, šalje zahtev za potvrdu validnosti skupa t-promenljivih nad kojima je potrebno obaviti ažuriranje, slika 4.2 *Faza 1*.

• Kada sakupi odgovore svih servera, sledi odlučivanje po sledećem principu: ukoliko je bar jedan od odgovora negativan, svim serverima se šalje naredba za prekid (*ABORT)*, u suprotnom šalje se naredba za ažuriranje (*COMMIT)*.

• Po prijemu naredbe, svaki server izvršava ažuriranje (COMMIT) ili odustaje (ABORT) i šalje povratnu informaciju koordinatoru o izvršenoj naredbi, slika 4.2 *Faza 2*. U ovom trenutku server koji je zatražio pokretanje 2PC protokola javlja korisniku, čiji je zahtev prihvatio, ishod i rezultat transakcije.

4.4 Model korisnika

Sa stanovišta servera, korisnik se može predstaviti kao automat koji šalje zahteve za obradu transakcije i od servera prima izveštaj o stanju, odnosno rezultat transakcije. Zarad ubrzanja simulacije automat korisnika je podešen tako da za svaku neuspelu transakciju započinje ponovni zahtev. Kada primi rezultat, u zavisnosti da li je transakcija uspešna ili neuspešna, korisnik će samo preuzeti rezultat ili zatražiti ponovnu obradu, respektivno. Pod ponovnom obradom podrazumeva se slanje zahteva za preuzimanje najnovijih verzija skupa t-promenljivih nad kojima korisnik operira. Po prihvatanju najnovijih verzija skupa t-promenljivih i unutrašnje obrade, korisnik šalje serveru zahtev za validaciju odnosno ažuriranje stanja t-promenljivih nad kojima je obavljao operacije.

Model korisnika DPSTM sistema implementiran je u klasi *DatacenterBroker.java*. Naziv klase (engl. *Data Center Broker*) u najjednostavnijem smislu opisuje ulogu korisnika, a to je da je korisnik posrednik servera, odnosno centra za obradu podataka.

4.5 Mrežne topologije

Mrežna topologija predstavlja strukturu povezanosti različitih elemenata komunikacione mreže, poput čvorova i veza [17]. Topološka struktura mreže može se prikazati fizički ili logički. Fizička topologija se odnosni na tip i raspored kablova koji se koristi za povezivanje uređaja, lokacije čvorova, kao i međusobne veze između čvorova i kablova [18]. Fizička topologija mreže je, takođe određena sposobnostima uređaja za pristup mreži i medijima, željenog nivoa kontrole

ili tolerancije greške. Nasuprot tome, logička topologija je način na koji signali deluju na mrežnom medijumu ili na koji način podaci prolaze kroz mrežu sa jednog uređaja na drugi bez obzira na fizičku međusobnu povezanost uređaja. Logička topologija mreže nije nužno ista kao i njegova fizička topologija. Logičke topologije su često blisko povezane sa metodama i protokolima kontrole pristupa medijima. Neke mreže su u mogučnosti da dinamički promene svoju logičku topologiju kroz promene konfiguracije u svojim usmerivačima i prekidačima (engl. *Router and Switshes*).

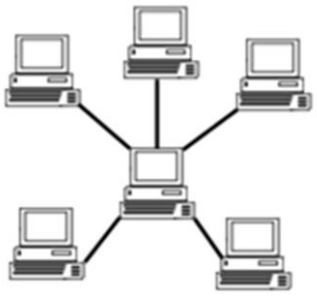
Razmak između čvorova, fizičkih međusobnih veza, brzine prenosa ili tipova signala može se razlikovati između dve mreže, ali njihove topologije mogu biti identične. Primer je lokalna mreža (engl. *Local Area Network, LAN*), gde svaki čvor ima jednu ili više fizičkih veza sa drugim uređajima u mreži, međutim mapiranje protoka podataka između komponenti određuje logičku topologiju mreže.

Modelovanje mrežne topologije odrađeno je kroz konfigurisanje *.brite* datoteke. Za potrebe testiranja simulacije napravljene su konfiguracione datoteke koje opisuju sledeće topologije:

- topologija magistrale
- topologija zvezde
- potpuno povezan graf
- nepotpuno povezan graf
- topologija klastera

4.5.1 Topologija zvezde

Model topologije zvezde korišćen je u ranoj fazi implementacije simuliranog modela DPSTM sistema. Centralni računar na Sl 4.3 predstavlja jedinstveni server u DPSTM sistemu.



Slika 4.3Topologija zvezde

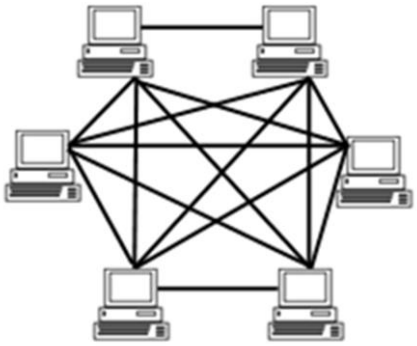
Klijenti, predstavljeni kao periferni računari, šalju zahteve za obradu transakcije.

Pošto se u sistemu nalazi samo jedan server, nadolazeći zahtevi poslati od strane korisnika obrađuju se sekvencijalno. Upotreba ove mreže koristi se za testiranje ispravnosti obrade transakcije na serveru, bez razmatranja brzine prenosa, kao ni vremena obrade.

4.5.2 Potpuno povezan graf

U potpuno povezanoj mreži (potpuno povezan graf), svi čvorovi su međusobno povezani, Sl 4.4. Međutim, pošto broj veza raste kvadratno sa brojem čvorova:

$$c = n(n - 1) / 2$$



Slika 4.4Potpuno povezan graf

gde je *c* broj veza, a *n* broj čvorova, ova topologija je nepraktična za velike mreže

Iz navedenog razloga topologija potpuno povezane mreže korišćena je u ranoj fazi ispitivanja korektnosti simuliranog modela DPSTM sistema sa uticajem mrežnog kašnjenja. Ovaj tip mreže koristi se i u ispitivanju i otklanjanju grešaka.

4.5.3 Nepotpuno povezan graf

U delimično povezanoj mreži određeni čvorovi su povezani na tačno jedan čvor, ali neki čvorovi su povezani sa dva ili više drugih čvorova direktnom vezom. Ovo omogućava korišćenje neke od redundancija mrežne topologije koja je fizički potpuno povezana, bez troškova i složenosti potrebne za vezu između svakog čvora u mreži.

Za mreže većih razmera, ovakva topologija predstavlja jedno od najboljih rešenja, pa je zbog toga i uzeta, i najviše razmatrana u simulaciji modela DPSTM sistema.

Svi rezultati prikazani u radu, gde je mreža uzeta u razmatranje, oslanjaju se na povezanost čvorova DPSTM sistema ovom topologijom.

4.5.4 Klaster

Računarski klaster (engl. *Cluster*) je skup usko povezanih računara koji rade zajedno tako da se mogu posmatrati kao jedan računar. Delovi klastera su obično spojeni brzom lokalnom

mrežom. Koristi se kako bi se povećale performanse u odnosu na one kod jednog računara, dok je jeftiniji u poređenju sa jednim računarom istih karakteristika.

Grid klaster je tehnologija usko povezana s klaster računarstvom. Glavna razlika između gridova i konvencionalnih klastera je to što gridovi obuhvataju skup računara koja ne veruju u potpunosti jedan drugom, i stoga rade manje kao jedan jedinstven računar. Takođe, gridovi obično podržavaju više heterogene skupove nego što su oni uobičajeni u klasterima.

Grid računarstvo namenjeno je za rad s opterećenjima koja se sastoje od više nezavisnih poslova (engl. *Jobs*) i paketa, kojima nije potrebna razmena podataka, jedni između drugih tokom procesa računanja. Gridovi služe za upravljanje poslovima na pojedinim računarima koji se izvršavaju nezavisno od ostatka grid-klastera. Resursi poput kapaciteta za skladištenje mogu biti deljeni na svim čvorovima, ali međurezultati ne utiču na izvršavanje poslova na odvojenim čvorovima.

4.5.5 Struktura .brite datoteke

Najjednostavniji način predstavljanja mrežne topologije unutar CloudSim simulatora jeste preko *.brite* datoteke. Pomoću ove datoteke moguća je konfiguracija cele mreže u sistemu, uključujući i fizičku i logičku topologiju mreže.

U *.brite* datoteci nalaze se podaci o modelu mreže, fizičkom rasporedu čvorova, međusobnoj udaljenosti, broju ulaza i izlaza jednog čvora, broju čvorova koji ga okružuje, itd. Međutim, drugi deo ove konfiguracione datoteke omogućuje povezivanje čvorova u željenu topologiju, unošenje mrežnog kašnjenja kao i brzine protoka za svaku liniju u mreži.

4.5.5.1 Primer magistrale (bus.brite)

Da bi se bolje razumela .brite datoteka, sledi primer konfiguracije za topologiju magistrale [14]. U mreži se nalaze šest čvora i deset veza koje ih povezuju. Pošto su linije jednosmerne mora se postaviti po jedna linija za svaki smer ukoliko se očekuje da komunikacija bude dvosmerna.

```
Topology: ( 6 Nodes, 10 Edges )
Model (1): 3 1000   100    1      1      2      0.15   0.2    1      10     1024

Nodes: ( 6 )
0    216    663    2      2      -1     RT_NONE
1    347    333    2      2      -1     RT_NONE
2    27     309    2      2      -1     RT_NONE
3    384    187    2      2      -1     RT_NONE
4    187    212    2      2      -1     RT_NONE
5    926    615    2      2      -1     RT_NONE
```

```
Edges: ( 10 )
0    0    1    355.05 1.18    10.0    -1    -1    E_RT_NONE
1    1    2    320.89 1.06    10.0    -1    -1    E_RT_NONE
2    2    3    377.27 1.25    10.0    -1    -1    E_RT_NONE
3    3    4    198.58 0.66    10.0    -1    -1    E_RT_NONE
4    4    5    841.74 2.80    10.0    -1    -1    E_RT_NONE
5    1    0    355.05 1.18    10.0    -1    -1    E_RT_NONE
6    2    1    320.89 1.06    10.0    -1    -1    E_RT_NONE
7    3    2    377.27 1.25    10.0    -1    -1    E_RT_NONE
8    4    3    198.58 0.66    10.0    -1    -1    E_RT_NONE
9    5    4    841.74 2.80    10.0    -1    -1    E_RT_NONE
```

Konfiguraciona datoteka se sastoji iz tri sekcije:

1. Model: informacije neophodne za generisanje topologije sadržane u datoteci, uključujući broj čvorova, broj veza, itd.

2. Nodes: svaki čvor u grafu predstavljen je jednom linijom u sledećem formatu:

NodeID xpos ypos indegree outdegree ASid type

TABELA 4.1

OBJAŠNJENJE POLJA IZLAZNE .BRITE DATOTEKE ZA OPIS ČVOROVA

Polje	Opis
NodeID	Unikatan identifikator za svaki čvor
xpos	Koordinata na x-osi u ravni
ypos	Koordinata na y-osi u ravni
indegree	Ulazni stepen čvora
outdegree	Izlazni stepen čvora
ASid	Identifikator Autonomnog Sistema (AS), kojem dati čvor pripada
type	Tip čvora

3. Edges: svaka veza u grafu predstavljena je jednom linijom u sledećem formatu: EdgeID from to length delay bandwith ASfrom ASto type

TABELA 4.2

OBJAŠNJENJE POLJA IZLAZNE .BRITE DATOTEKE ZA OPIS VEZA

Polje	Opis
EdgeID	Unikatan identifikator za svaku vezu
from	ID izvorišnog čvora
to	ID odredišnog čvora
length	Euklidska dužina
delay	Kašnjenje
bandwidth	Protok
ASfrom	ID AS izvoričnog čvora
ASto	ID AS odredišnog čvora
type	Tip

5. Programska implementacija

Unutar ovog odeljka objašnjeni su detalji implementacije za svaki od navedenih modula pomenutih u prethodnom poglavlju. U procesu implementacije proširene su već postojeće CloudSim klase sa malim izmenama originalne implementacije.

U tabeli 1 prikazan je spisak svih modula korišćenih pri implementaciji programskog rešenja. Uz svaki modul dat je kratak opis njegove funkcionalnosti. U daljem delu poglavlja biće opisani samo najvažniji deo prigramske sprege (engl. *Application Interface, API*) modula koji omogućavaju simulaciju modela DPSTM sistema.

TABELA 5.1

SPISAK MODULA NEOPHODNIH ZA SIMULACIJU DSTM

Naziv modula	Datoteke modula (.java)	Kratak opis modula
T-variable	Tvar.java	Objekat transakcione promenljive
Transaction	Transaction.java	Objekat transakcije. Omogućava rad sa transakcionim promenljivama.
TransactionDatacenterBroker	TransactionDatacenterBroker.java	Modul koji predstavlja krajnjeg korisnika.
TransactionDatacenter	TransactionDatacenter.java	Modul predstavlja centar za obradu podataka na kome se

		transakcije izvršavaju
Cloudlet	Cloudlet.java	Objekat koji modeluje usluge aplikacija zasnovane na oblaku
CloudInformationService	CloudInformationService.java	Usluga informacije o oblaku. Pruža usluge registrovanja resursa u oblaku, indeksiranja i usluga otkrivanja. Ukratko, ponaša se kao usluga žuth stranica (telefonski imenik).
CloudSimTags	CloudSimTags.java	Sadrži različite statičke oznake za naredbe koje ukazuju na vrstu akcije koju CloudSim entiteti moraju preduzeti prilikom primanja ili slanja poruke događaja.
CloudSimShutdown	CloudSimShutdown.java	Čeka završetak svih CloudSim korisničkih entiteta kako bi odredio kraj simulacije.

5.1 Modul Tvar

Tvar modul implementiran u klasi Tvar.class predstavlja objekat transakcione promenljive, koja čini osnovni gradivni element transakcije. Upravljanje transakcionim promenljivama omogućeno je kroz API koji čine sledeće metode:

- Tvar(String)
- Tvar(String, int)
- int getVersion(void
- void setVersion(int)
- String getID(void)

5.1.1 Konstruktor Tvar

Opis: Pravi objekat transakcione promenljive

Prototip: Tvar(String key, int version)

Ulazni argumenti:

1. key – ključ/jedinstven naziv transakcione promenljive
2. version – trenutna verzija transakcione promenljive

Povratna vrednost: kod greške

5.1.2 Metoda getVersion

Opis: Dobavlja trenutnu verziju transakcione promenljive

Prototip: int getVersion(void)

Ulazni argumenti: nema

Povratna vrednost: Trenutna verzja transakcione promenljve

5.1.3 Metoda setVersion

Opis: Postavlja verziju transakcione promenljive na željenu vrednost

Prototip: void setVersion(int version)

Ulazni argumenti:

1. version – verzija koja se dodeljuje transakcionoj promenljivoj

Povratna vrednost: kod greške

5.1.4 Metoda getID

Opis: Dobavlja ključ, odnosno jedinstven naziv transakcione promenljive

Prototip: String getID(void)

Ulazni argumenti: nema

Povratna vrednost: Ključ transakcione promenljive

5.2 Modul Transaction

Klasa *Transaction* sadrži implementaciju prethodno pomenutog modela transakcije. Objekat ove klase je sastavni deo poruke događaja koju broker šanje VM i obrnuto. Transakcija može, kao što je već pomenuto, da izvršava operacije čitanja i/ili pisanja. Upravljanje transakcijom omogućeno je kroz Transacion API koji čine sledeće metode:

- Transaction(int, long, int, long, long, UtilizationModel, UtilizationModel, UtilizationModel, boolean, List<String>)
- int getWriteOpDuration(void)
- int getReadOpDuration(void)
- <T extends Tvar> void setRead(List<T> read)
- <T extends Tvar> List<T> getWrite(void)
- <T extends Tvar> void setWrite(List<T> write)
- <T extends Tvar> List<T> getRead(void)

• int getType(void)

• void setType(int type)

• int getTransactionStatus(void)

• void setTransactionStatus(int transactionStatus)

• setFinish(boolean val)

5.2.1 Konstruktor Transacion

Opis: Pravi objekat transakcije

Prototip: public Transaction(int cloudletId, long cloudletLength, int pesNumber,
long cloudletFileSize, long cloudletOutputSize,
UtilizationModel utilizationModelCpu,
UtilizationModel utilizationModelRam,
UtilizationModel utilizationModelBw, boolean record,
List<String> fileList)

Ulazni argumenti:

1. int transactionId – jedinstveni ključ objekta transakcije
2. long transactionLength – veličina transakcije izražena u MI
3. int pesNumber – broj procesorskih jedinica neophodnih za obradu transakije
4. long transactionFileSize – veličina transakcije pre obrade izražena u MB
5. long transactionOutputSize – veličina transakcije posle obrade izražena u MB
6. UtilizationModel utilizationModelCpu – model korišćenja CPU
7. UtilizationModel utilizationModelRam – model korišćenja RAM
8. UtilizationModel utilizationModelBw – model korišćenja propusnog opsega
9. boolean record – pamtiti istoriju objekta ili ne
10. List<String> fileList – spisak fatoteka potrebnih ovoj transakciji

Povratna vrednost: Objekat transakcije

5.2.2 Metoda getReadOpDuration

Opis: Dobavlja broj transakcionih promenljivih nad kojima se obavlja operacija čitanja

Prototip: int getReadOpDuration(void)

Ulazni argumenti: nema

Povratna vrednost: Broj transakcionih promenljivih nad kojima se obavlja operacija čitanja

5.2.3 Metoda getWriteOpDuration

Opis: Dobavlja broj transakcionih promenljivih nad kojima se obavlja operacija pisanja

Prototip: int getWriteOpDuration(void)

Ulazni argumenti: nema

Povratna vrednost: Broj transakcionih promenljivih nad kojima se obavlja operacija pisanja

5.2.4 Metoda setRead

Opis: Transakciji dodeljuje listnu transakcionih promenljivih nad kojima se obavlja operacija čitanja

Prototip: <T extends Tvar> void setRead(List<T> read)

Ulazni argumenti:

1. read – lista transakcionih promenljivih

Povratna vrednost: nema

5.2.5 Metoda setWrite

Opis: Transakciji dodeljuje listu transakcionih promenljivih nad kojima se obavlja operacija pisanja

Prototip: <T extends Tvar> void setWrite(List<T> write)

Ulazni argumenti:

1. write – lista transakcionih promenljivih

Povratna vrednost: nema

5.2.6 Metoda getRead

Opis: Dobavlja listu transakcionih promenljivih nad kojima se obavlja operacija čitanja

Prototip: <T extends Tvar> List<T> getRead(void)

Ulazni argumenti: nema

Povratna vrednost: Lista transakcionih promenljivih

5.2.7 Metoda getWrite

Opis: Dobavlja listu transakcionih promenljivih nad kojima se obavlja operacija pisanja

Prototip: <T extends Tvar> List<T> getWrite(void)

Ulazni argumenti: nema

Povratna vrednost: Lista transakcionih promenljivih

5.2.8 Metoda setType

Opis: Određuje tip transakcije

Prototip: void setType(int type)

Ulazni argumenti:

1. type – tip transakcije: GET_READ, GET_WRITE, GET_R_W, COMMIT, COMMIT_VAL, COMMIT_R_W

Povratna vrednost: nema

5.2.9 Metoda getType

Opis: Dobavlja tip transakcije

Prototip: int getType(void)

Ulazni argumenti: nema

Povratna vrednost: Tip transakcije: GET_READ, GET_WRITE, GET_R_W, COMMIT, COMMIT_VAL, COMMIT_R_W

5.2.10 Metoda getTransactionStatus

Opis: Dobavlja status tranakcije, odnosno da li je transakcija gotova ili ne

Prototip: int getTransactionStatus(void)

Ulazni argumenti: nema

Povratna vrednost: Status transakcije

5.2.11 Metoda setTransactionStatus

Opis: Postavlja status izvršavanja transakcije

Prototip: void setTransactionStatus(int transactionStatus)

Ulazni argumenti:

1. transactionStatus – trenutni status transakcije

Povratna vrednost: nema

5.2.12 Metoda setFinish

Opis: Postavlja promenljivu koja označava kraj transakcije na željenu vrednost

Prototip: void setIsFinish(boolean val)

Ulazni argumenti:

1. val – kod završetka transakcije

Povratna vrednost: nema

5.3 Modul TransactionDatacenter

Klasa TransactionDatacenter sadrži implementaciju prethodno pomenutog modela centra za obradu podataka. Objekat ove klase sadrži podatke o VM na kojima se obrađuju transakcije. Upravljanje centrom za obradu podataka omogućeno je kroz TransactionDatacenter API koji čine sledeće metode:

- TransactionDatacenter(String, DatacenterCharacteristics,
  VmAllocationPolicy, List<Storage>, double)

- int getDictionarySize()
- String getDictionary()
- void setDatacetersID(List<Integer> idList, int self)
- List<Integer> getDatacetersID()
- void removeSelfID(int self)
- List<Integer> getIDList()
- void setIDList(List<Integer> iDList)
- void removeFromIDList(int self)
- boolean commitVars(List<Tvar> lst)
- boolean cmpVars(List<Tvar> lst)
- void abortVars(List<Tvar> lst)
- void putVars(List<Tvar> lst)
- List<Tvar> getVars(List<String> glst)
- void addVars(List<Tvar> lst)
- void executeTransaction(SimEvent ev)
- void processTransaction(SimEvent ev)
- void updateTransactionProcessing()
- void coordinator(SimEvent ev)
- void sendPrepareRequest(SimEvent ev)
- void preparePhase(SimEvent ev)
- void collectAnswers(SimEvent ev, boolean response)
- void commitPhase(SimEvent ev, boolean commit)
- void finalPhase(SimEvent ev)

5.3.1 Konstruktor TransactionDatacenter

Opis: Kreira objekat centra za obradu podataka

Prototip: TransactionDatacenter(String name, DatacenterCharacteristics c haracteristics, VmAllocationPolicy vmAllocationPolicy, List<Storage> storageList, double schedulingInterval)

Ulazni argumenti:

1. name – Ime centra za obradu podataka
2. characteristics – Karakteristike (PES, RAM, BW...)
3. vmAllocationPolicy – Polisa raspoređivanja VM
4. storageList – Spisak elemenata za skladištenje

5. schedulingInterval – Interval raspoređivanja

Povratna vrednost: TransactionDatacenter objekat

5.3.2 Metoda getDictionarySize

Opis: Dobavlja veličinu rečnika

Prototip: int getDictionarySize(void)

Ulazni argumenti: nema

Povratna vrednost: Veličina rečnika / transakcione memorije

5.3.3 Metoda getDictionary

Opis: Dobavlja rečnik / transakcionu memoriju centra za obradu podataka

Prototip: String getDictionary(void)

Ulazni argumenti: nema

Povratna vrednost: Rečnik / transakciona memorija centra za obradu podataka

5.3.4 Metoda setDatacetersID

Opis: Popunjava listu adresa svih centara za obradu podataka u sistemu

Prototip: void setDatacetersID(List<Integer> idList, int self)

Ulazni argumenti:

1. idList – lista adresa svih centara za obradu podataka
2. self – sopstvena adresa

Povratna vrednost: nema

5.3.5 Metoda getDatacetersID

Opis: Dobavlja listu adresa svih distribuiranih centara za obradu podataka u sistemu

Prototip: List<Integer> getDatacetersID(void)

Ulazni argumenti: nema

Povratna vrednost: Lista adresa distribuiranih centara za obradu podataka

5.3.6 Metoda removeSelfID

Opis: Briše sopstvenu adresu iz liste adresa svih distribuiranih centara za obradu podataka u sistemu

Prototip: void removeSelfID(int selfID)

Ulazni argumenti:

1. selfID – sopstvena adresa

Povratna vrednost: nema

5.3.7 Metoda commitVars

Opis: Dobavlja rečnik / transakcionu memoriju centra za obradu podataka

Prototip: boolean commitVars(List<Tvar> lstTvar)

Ulazni argumenti:

1. lstTvar – lista transakcionih promenljivih

Povratna vrednost: Kod greške

5.3.8 Metoda cmpVars

Opis: Upoređuje transakcione promenljive sa stanjem u rečniku / TM

Prototip: boolean cmpVars(List<Tvar> lst)

Ulazni argumenti:

1. lst – lista transakcionih promenljivih

Povratna vrednost: kod greške

5.3.9 Metoda abortVars

Opis: Poništavanje prethodno obavljenog ažuriranja rečnika / TM

Prototip: void abortVars(List<Tvar> lst)

Ulazni argumenti:

1. lst – lista transakcionih promenljivih

Povratna vrednost: nema

5.3.10 Metoda putVars

Opis: Nekontrolisano dodavanje i ažuriranje rečnika / transakcione memorije

Prototip: void putVars(List<Tvar> lst)

Ulazni argumenti:

1. lst – lista transakcionih promenljivih koji se ubacuju u TM

Povratna vrednost: nema

5.3.11 Metoda getVars

Opis: Dobavlja listu transakcionih promenljivih iz rečnika / transakione memorije

Prototip: List<Tvar> getVars(List<String> lst)

Ulazni argumenti:

1. lst – lista ključeva transakcionih promenljivih koje treba dobaviti

Povratna vrednost: lista transakcionih promenljivih

5.3.12 Metoda addVars

Opis: Dodaje transakcione promenljive u rečnik / transakcionu memoriju

Prototip: void addVars(List<Tvar> lst)

Ulazni argumenti:

1. lst – lista transakcionih promenljivih koja se dodaje u rečnik

Povratna vrednost: nema

5.3.13 Metoda executeTransaction

Opis: Pokreće izvršavanje transakcije

Prototip:void executeTransaction(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.3.14 Metoda processTransaction

Opis: Procesira transakciju

Prototip: void processTransaction(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.3.15 Metoda updateTransactionProcessing

Opis: Ažurira stanje transakcije tokom njene obrade na centru za obradu podataka, odnosno VM

Prototip: void updateTransactionProcessing(void)

Ulazni argumenti: nema

Povratna vrednost: nema

5.3.16 Metoda coordinator

Opis: Prihvata zahtev za pokretanje 2PC protokola.

Prototip: void coorinator(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.3.17 Metoda sendPrepareRequest

Opis: Pokreće 2PC protokol, radi korektnog ažuriranja TM

Prototip: void sendPrepareRequest(SimEvent ev)

Ulazni argumenti:

2. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.3.18 Metoda preparePhase

Opis: Proverava da li je moguće lokalno obraditi zahtev primljen preko 2PC protokola

Prototip: void preparePhase(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.3.19 Metoda collectAnswers

Opis: Sakuplja odgovore prve faze 2PC protokola

Prototip: void collectAnswers(SimEvent ev, boolean response)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji
2. response – odgovor koji može biti pozitivan ili negativan

Povratna vrednost:

5.3.20 Metoda commitPhase

Opis: Ažurira stanje TM po zahtevu primljenom preko 2PC protokola

Prototip: void commitPhase(SimEvent ev, boolean commit)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji
2. commit – dozvola za obradom zahteva, može bti pozitivna ili negativna

Povratna vrednost:

5.3.21 Metoda finalPhase

Opis: Poslednja faza obrade transakcije. Po primanju svih pozitivnih odgovora putem 2PC protokola, ažurira se stanje TM

Prototip: void finalPhase(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja, u sebi sadrži podatke o transakciji

Povratna vrednost: nema

5.4 Modul TransactionDatacenterBroker

Klasa TransactionDatacenterBroker sadrži implementaciju prethodno pomenutog modela krajnjeg korisnika. Komunikacija između krajnjeg korisnika i centra za obradu podataka omogućeno je kroz TransactionDatacenterBroker API koji čine sledeće metode:

- TransactionDatacenterBroker(String)
- void submitTransactionList(List<? extends Transaction>)
- <T extends Transaction> List<T> getTransactionList(void)
- <T extends Transaction> void setTransactionList(List<T>)
- <T extends Transaction> List<T> getTransactionSubmittedList(void)
- <T extends Transaction> void setTransactionSubmittedList(List<T>)
- <T extends Transaction> List<T> getTransactionReceivedList(void)
- <T extends Transaction> List<T> getTransactionReceivedList(void)
- void submitTransactions(void)
- void submitTransactions(double)
- void processVmCreate(SimEvent)
- void processTransactionReturn(SimEvent)
- void bindTransactionToVm(int, int)
- void processOtherEvent(SimEvent)
- boolean checkResult(SimEvent)

5.4.1 Konstruktor TransactionDatacenterBroker

Opis: Kreira objekat krajnjeg korisnika

Prototip: TransactionDatacenterBroker(String name)

Ulazni argumenti:

1. name – Ime brokera / krajnjeg korisnika

Povratna vrednost: Objekat TransactionDatacenterBroker

5.4.2 Metoda submitTransactionList

Opis: Služi kako bi se lista transakcija poslala brokeru

Prototip:void submitTransactionList(List<? extends Transaction> list)

Ulazni argumenti:

1. list – lista transakcija

Povratna vrednost: nema

5.4.3 Metoda getTransactionList

Opis: Dobavlja listu transakcija

Prototip:<T extends Transaction> List<T> getTransactionList(void)

Ulazni argumenti: nema

Povratna vrednost: lista transakcija

5.4.4 Metoda setTransactionList

Opis: Popunjava listu transakcija

Prototip:<T extends Transaction> void setTransactionList(List<T> transactionList)

Ulazni argumenti:

1. transactionList – nova lista transakcija

Povratna vrednost: nema

5.4.5 Metoda getTransactionSubmittedList

Opis: Dobavlja podnetu listu transakcija

Prototip:<T extends Transaction> List<T> getTransactionSubmittedList(void)

Ulazni argumenti: nema

Povratna vrednost: lista transakcija

5.4.6 Metoda setTransactionSubmittedList

Opis: Popunjava listu transakcija koja se podnosi/šalje na obradu

Prototip: <T extends Transaction> void setTransactionSubmittedList(List<T> transactionSubmittedList)

Ulazni argumenti:

1. transactionSubmittedList – lista transakcija

Povratna vrednost: nema

5.4.7 Metoda getTransactionReceivedList

Opis: Dobavlja listu obrađenih transakcija

Prototip: <T extends Transaction> List<T> getTransactionReceivedList(void)

Ulazni argumenti: nema

Povratna vrednost: lista obrađenih transakcija

5.4.8 Metoda setTransactionReceivedList

Opis: Popunjava listu obrađenih transakcija

Prototip: <T extends Transaction> void setTransactionReceivedList(List<T> transactionReceivedList)

Ulazni argumenti:

1. transactionReceivedList – lista obrađenih transakcija

Povratna vrednost: nema

5.4.9 Metoda submitTransactions

Opis: Podnosi transakcije na prethodno napravljenu VM

Prototip:

1. void submitTransactions(void)
2. void submitTransactions(double delay)

Ulazni argumenti:

1. delay – željeno kašnjenje podnošenja zahteva

Povratna vrednost: nema

5.4.10 Metoda processVmCreate

Opis: Procesira odgovor centra za obradu podataka na zahtev za kreiranje VM

Prototip: void processVmCreate(SimEvent ev)

Ulazni argumenti:

1. ev - objekat događaja

Povratna vrednost: nema

5.4.11 Metoda processTransactionReturn

Opis: Procesira događaj povratka transakcije, odnosno završetka obrade transakcije

Prototip: void processTransactionReturn(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja

Povratna vrednost: nema

5.4.12 Metoda bindTransactionToVm

Opis: Specificira obradu željene transakcije na tačno određenoj VM

Prototip: void bindTransactionToVm(int transationId, int vmId)

Ulazni argumenti:

1. transationId – ključ transakcije, po kojem se može identifikovati
2. vmId – ključ VM, po kojem se može identifikovati

Povratna vrednost: nema

5.4.13 Metoda processOtherEvent

Opis: Služi za proširenje, odnosno obradu svih nepoznatih događaja

Prototip: void processOtherEvent(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja

Povratna vrednost: nema

5.4.14 Metoda checkResult

Opis: Proverava rezultat primljene transakcije. Ukoliko su samo dobavljene željene transakcione promenljive, tada se prelazi na ažuriranje (commit), odnosno proveru validnosti (commit_validate).

Prototip: boolean checkResult(SimEvent ev)

Ulazni argumenti:

1. ev – objekat događaja

Povratna vrednost: rezultat obrade transakcije. Može biti pozitivan ili negativan

6. Evaluacija

Evaluacija sistema podeljena je u dve etape. Prva je analiza i testiranje sistema u idelanim uslovima, gde se akcenat stavlja samo na ispravnost algoritma obrade transakcije u distribuiranom okruženju i na komunikaciju centara za obradu podataka prilikom obrade transakcije. U okviru druge etape akcenat se prebacuje na analizu rada implementiranog modela DPSTM sistema u što je realnijim uslovima moguće.

Kako se radi o simulaciji distribuirane pajton softverske transakcione memorije, evaluacija i testiranje obavljeno je kroz analizu i upoređivanje dobijenih sa prethodno izvedenim teorijskim rezultatima. Testni scenariji počinju od najjednostavnijih slučajeva, pa do onih kritičnih koji bi mogli uticati i narušiti ispravnost rada sistema.

6.1 Evaluacija bez modela mreže

Kao što je prethodno pomenuto, prvo je testirana ispravnost rada sistema u idealnim uslovima. Pod idealnim uslovom podrazumeva se da je mreža idealna – nema kašnjenja, brzina prenosa i obrade transakcije je velika i ne zavisi od broja t-promenljivih nad kojima se transakcija odvija. Kako se ne uzima u razmatranje mreža, podrazumeva se da su svi čvorovi, centri za obradu podataka i klijenti koji šalju zahteve, potpuno povezani i podjednako udaljeni.

Nakon analize algoritma obrade transakcije na centru za obradu podataka, uvodi se zavisnost broja t-promenljivih nad kojima se transakcija obavlja i dužine trajanja transakcije. Pored ove, postoji i zavisnost dužine trajanja transakcije od tipa operacije (READ, WRITE) koja se obavlja nad određenom t-promenljivom, koja se, takođe, uzima u obzir prilikom analize ispravnosti rada sistema.

6.2 Evaluacija sa modelom mreže

Druga etapa evaluacije predstavlja analizu i testiranje simuliranog modela DPSTM sistema u realnim uslovima. Nakon utvrđenog ispravnog rada sistema, kada su svi čvorovi u mreži potpuno povezani i podjednako udaljeni, analizira se uticaj promene mrežne topologije na rad sistema. Važno je napomenuti da se do sada svaka transakcija smatrala kao jedinstven paket iste veličine, bez obzira na broj t-promenljivih nad kojima operiše.

Za ispitivanje, modelovane su topologija magistrale, zvezde, nepotpunog i potpuno povezanog grafa. Pored fizičke povezanosti čvorova, uvedena su i razna mrežna kašnjenja. Upoređivanjem dobijenh rezultata, za svaku od navedenih mrežnih topologija, verifikovana je ispravnost rada simuliranog modela DPSTM sistema. Na samom kraju uvedena je zavisnost veličine transakcije, odnosno paketa koji se šalje kroz mrežu, od broja t-promenljivih nad kojima se transakcija obavlja. Ova zavisnost uvedena je radi dobijanja realističnijh testnih scenarija i rezultata koji bolje opisuju ponašanje opisanog sistema, odnosno DPSTM.

Upoređivanjem dobijenih rezultata pokazano je da performanse sistema, odnosno brzina obrade transakcije, zavise ne samo od broja t-promenljivih nad kojima se transakcija obavlja, nego i od međusobne povezanosti i udaljenosti čvorova u sistemu. Ključnu ulogu ima mrežno kašnjenje, koje može promeniti redosled obrade transakcija, odnosno toka događaja u sistemu.

7. Rezultati

U ovom odeljku dati su rezultati simuliranih relevantnih slučajeva izvršavanja. Rezultati su, kao i u prethodnom poglavlju, radi lakše predstave i bolje preglednosti predstavljeni u dva potpoglavlja:

- bez modelovane mrežne topologije
- sa modelovanom mrežnom topologijom

7.1 Rezultati testiranja sistema bez modelovane mrežne topologije

Tabela 7.1 sadrži rezultate grupe transakcija koje obavljaju operaciju čitanja. Sve transakcije se izvršavaju na istom centru za obradu podataka, pri čemu svaka od njih sadrži bar jednu deljenu transakcionu promenljivu. Po vremenu početka *Start* i vremenu trajanja transakcije *Trajanje*, može se videti da se transakcije izvršavaju paralelno. Vremena u tabelama data su u neimenovanoj jedinici. Pošto se izvršavaju samo transakcije koje obavljaju operaciju čitanja, ni jedna transakcija nije u konfliktu i sve se izvršavaju uspešno.

TABELA 7.1

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU OPERACIJU ČITANJA NA JEDNOM SERVERU

Start	Trajanje	Čitanje/Upis	Neuspešno
0.1	1600.1	5/-	0
0.1	1600.1	5/-	0
0.1	1600.1	5/-	0
0.1	1600.1	5/-	0
0.1	1600.1	5/-	0

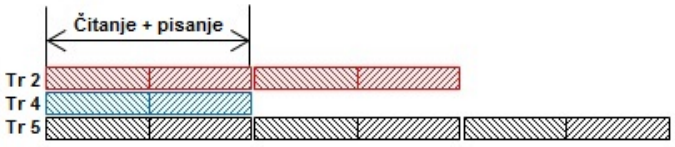
Vreme trajanja transakcije u simulaciji preračunava se na osnovu definisanih karakteristika procesa, odnosno transakcije. Svaka transakcija defnisana je, kao što je već pomenuto, neophodnom RAM memorijom i memorijom za skladištenje, kao i brojem instrukcija neophodnim da bi se izvršila. Broj instrukcija izražava se u jedinici MI, milion instrukcija (engl. *Million Instruction*). Virtuelna mašina (VM) definisana je brojem procesorskih jedinica, gde je svaka procesorska jedinica definisana jedinicom MI [19]. Na osnovu karakteristika VM i definisanog procesa, odnosno transakcije, CloudSim automatski preračunava vreme potrebno da se dati proces, odnosno transakcija izvrši.

TABELA 7.2

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU OPERACIJU ČITANJA ODNOSNO PISANJA NA JEDNOM SERVERU

ID	Start	Trajanje	Tip	Neuspešno
TR1	0.1	1600.1	Čitanje	0
TR3	0.1	1600.1	Čitanje	0
TR4	0.1	1600.1	Upis	0
TR2	1600.1	3200.1	Upis	1
TR5	3200.1	4800.1	Upis	2

U Tabeli 7.2 dati su rezultati grupe transakcija koje se izvršavaju na jednom serveru, odnosno centru za obradu podataka. Sve transakcije se pokreću u isto vreme. Transakcije TR1 i TR3 obavljaju operaciju čitanja, dok transakcije TR2, TR4 i TR5 obavljaju operaciju pisanja. Pošto sve transakcije rade nad istim skupom transakcionih promenljivih, dolazi do konflikta među transakcijama koje obavljaju operaciju pisanja. Među transakcijama koje su u konfliktu, samo jedna može uspešno da se izvrši u datom momentu, dok se ostale izvršavaju neuspešno. U ovom slučaju uspešno se izvršila transakcija TR4 i ona se izvršava peralelno sa transakcijama koje obavljaju operaciju čitanja TR1 i TR3. Vremena početka i trajanja transakcija TR2 i TR5 posledica su obavljanja operacije čitanja pre ponovne operacije pisanja. Svakim neuspešnim ažuriranjem transakcija izvršava operaciju čitanja nad skupom transakcionih promenljivih koje želi da ažurira, kako bi dobavila najnovije verzije istih.



Slika 7.11Prikaz izvršavanja transakcija u vremenu

Na slici 7.1 dat je grafički prikaz izvršavanja transakcija TR2, TR4 i TR5 iz Tabele 7.2. Posle svake neuspešno završene transakcije moraju se dobaviti najnovije verzije za skup transakcionih promenljivih nad kojima transakcija obavlja operacije pisanja, odnosno želi da ažurira. Kao što se i sa grafika može videti, vreme trajanja transakcije direktno je proporcionalno broju prethodno neuspešnih izvršavanja.

TABELA 7.3

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU OPERACIJU ČITANJA I PISANJA NA RAZLIČITIM SERVERIMA

ID	Start	Trajanje	Čitanje/Upis	Neuspešno
TR1	0.1	1600.1	5/5	0
TR2	1600.2	3200.2	5/5	1
TR3	3200.2	4800.3	5/5	2
TR4	4800.3	6400.4	5/5	3
TR5	6400.4	8000.5	5/5	4

U Tabeli 7.3 prikazani su rezultati za grupu od pet transakcija koje obavljaju i čitanje i pisanje, svaka na različitom centru za obradu podataka. Sve transakcije koriste isti skup t-promenljivih i počinju u približno isto vreme. Svaka od transakcija pokušava da ažurira prethodno čitanje t-promenljive, pa su stoga sve transakcije u konfliktu. Ovaj konflikt utiče na vreme izvršavanja transakcija. Vreme izvršavanja druge najkraće transakcije duplo je veće od vremena izvršavanja prve, što je posledica prvog neuspešnog ažuriranja.

Nakon utvrđivanja ispravnosti rada, odnosno verifikacije ispravnosti algoritma obrade i komunikacije između centara za obradu podataka, obavljena je korekcija parametara koji definišu transakciju. Uvedena je pomenuta zavisnost vremena izvršavanja transakcije od broja t-promenljivih i tipa operacije (*READ, WRITE*) koja se nad njima obavlja.

TABELA 7.4

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU VALIDACIJU NA RAZLIČITIM SERVERIMA

ID	Početak Dobavljanja	Početak Validacije	Kraj	Trajanje	Validacija / Ažuriranje	Neuspešno
TR1	0.1	4.1	12.1	12	10/0	0
TR2	0.2	4.2	12.2	12	10/0	0
TR3	0.3	4.3	12.3	12	10/0	0
TR4	0.4	4.4	12.4	12	10/0	0
TR5	0.5	4.5	12.5	12	10/0	0

U tabeli 7.4 dati su rezultati za grupu od pet transakcija. Svaka transakcija obavlja validaciju skupa od deset deljenih t-promenljivih na zasebnom serveru, odnosno centru za

obradu podataka. Razlika u vremenu početka transakcija rezultat je diskretizacije događaja u CloudSim simulatoru. Ako se ta razlika zanemari, može se reći da transakcije iz tabele 7.4 počinju i završavaju u približno isto vreme, odnosno da se izvršavaju u paraleli.

TABELA 7.5

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU AŽURIRANJE NA RAZLIČITIM SERVERIMA

ID	Početak Dobavljanja	Početak Ažuriranja	Kraj	Trajanje	Validacija / Ažuriranje	Neuspešno
TR1	0.1	5.1	13.9	13.8	0/10	0
TR2	13.9	19.4	28.2	14.3	0/10	1
TR3	28.2	33.7	42.5	14.3	0/10	2
TR4	42.5	48	56.8	14.3	0/10	3
TR5	56.8	62.3	71.1	14.3	0/10	4

Rezultati za grupu od pet transakcija koje obavljaju ažuriranje deset deljenih t-promenljivih, dati su u tabeli 7.5. Svaka od transakcija se obavlja na zasebnom serveru, odnosno centru za obradu podataka, kao što je to bio slučaj i u prethodnom primeru.

Kolona *Početak Dobavljanja* predstavlja vreme početka, dok kolona *Kraj* predstavlja vreme završetka uspešno obavljene transakcije. Iz tabele 7.5 se može videti da je razlika u vremenima kolona *Početak Dobavljanja* i *Početak Ažuriranja* za većinu jednaka 5.5 neimenovanih vremenskih jedinica. To vreme jednako je vremenu potrebnom da se dobavi željeni skup od deset t-promenljivih nad kojima se radi ažuriranje, tj operacija čitanja za skup od deset t-promenljivih traje toliko.

Još jedno opažanje se može uočiti, a to je vreme trajanja transakcija koje se znatno razlikuje između tabela 7.4 i 7.5 i tabela 7.1, 7.2 i 7.3. Iako se u tabelama 7.4 i 7.5 obavlja validacija, odnosno ažuriranje za skup od 10 t-promenljivih, što je duplo više nego u prethodne tri tabele, vreme trajanja transakcije je znatno manje. Razlog ovome je već pomenuto korigovano vreme izvršavanja koje zavisi od broja t-promenljivih nad kojima se transakcija obavlja, kao i od tipa operacija (*READ, WRITE*) koja se obavlja nad određenim t-promenljivama.

7.2 Rezultati testiranja sistema sa modelovanom mrežnom topologijom

Uvođenjem modela mrežne topologije u smislu fizičke međusobne povezanosti čvorova i mrežnog kašnjenja između čvorova, dolazi do promene u performansi sistema. Ova promena se ogleda u vremenu potrebnom za slanje zahteva za obradu transakcije centru za obradu odataka, kao i međusobne komunikacije centara za obradu podataka. U zavisnosti od vremena potrebnog da korisnik svoj zahtev za obradu transakcije pošalje centru za obradu podataka menja se redosled izvršavanja transakcija.

Tabela 7.6 sadrži rezultate za grupu od pet transakcija poslatih od strane različitih korisnika na različite centre za obradu podataka. Svaka od transakcija obavlja validaciju za isti skup od deset deljenih t-promenljivih. Razlike u kolonama *Početak Validacije* i *Kraj* tabele 7.4 i *Prihvat Zahteva* tabele 7.6 posledica je uvedene topologije i mrežnog kašnjenja.

TABELA 7.6

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU VALIDACIJU NA RAZLIČITIM SERVERIMA

ID	Početak Dobavljanja	Početak Validacije	Kraj	Trajanje	Validacija / Ažuriranje	Neuspešno
TR1	3.35	7.35	11.35	8	10/0	0
TR2	0.95	8.95	12.95	12	10/0	0
TR3	2.05	10.05	14.05	12	10/0	0
TR4	3.65	11.65	15.65	12	10/0	0
TR5	1.75	13.75	17.75	16	10/0	0

Tabela 7.7 sadrži rezultate grupe transakcija koje obavljaju ažuriranje. Svaka od transakcija se izvršava na različitom centru za obradu podataka, pri čemu svaka od njih pokušava da ažurira isti skup t-promenljivih, što izaziva konflikt među svim transakcijama. Samo prva, najbrže obrađena transakcija će obaviti uspešno ažuriranje, dok će ostale biti neuspešne.

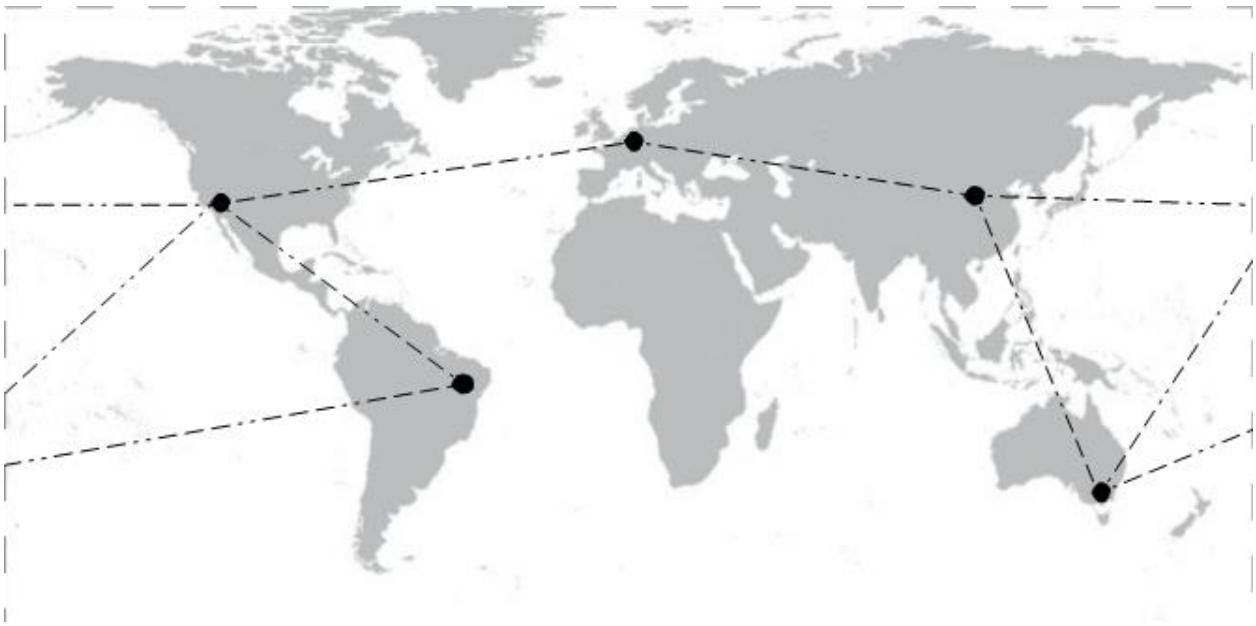
TABELA 7.7

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU AŽURIRANJE NA RAZLIČITIM SERVERIMA

ID	Prihvat I Zahteva	Početak Ažuriranja	Kraj	Trajanje	Validacija / Ažuriranje	Neuspešno
TR1	1.85	7.75	16.55	8.8	0/10	0
TR2	3.45	24.55	33.35	8.8	0/10	1
TR3	4.55	52.45	51.25	8.8	0/10	2
TR4	6.15	59.75	68.55	8.8	0/10	3
TR5	7.65	96.05	140.05	8.8	0/10	4

Kao i u prethodnom primeru, vremena u kolonama *Početak* i *Kraj* tabele 7.7 razliku se od vremena u tabeli 7.5. Međutim, još jedna razlika se može uočiti. Razlika u vremenu kraja jedne i početka druge uspešne transakcije nije uvek ista, odnosno nije ista za svaku od transakcija. Ovo je posledica toga što se svaka transakcija izvršava na zasebnom centru, a korisnici koji zahtev za obradu transakcije šalju centrima za obradu podataka, su nejednako udaljeni od centara, što utiče na vreme potrebno da se zahtev pošalje, odnosno primi rezultat obrade.

Slika 7.2 prikazuje fizičku povezanost čvorova u mreži korišćenu za prikaz rezultata u ovom radu. Uzeto je da se svaki od centara za obradu podataka nalazi na zasebnom kontinentu. Korisnici zahtev za obradu transakcije šalju centrima na kontinentu na kojem se i oni nalaze.



Slika 7.2Fizička povezanost čvorova u DSTM sistemu

Iako se može pretpostaviti da će se prvo izvršiti transakcija korisnika, koji je najbliži centru kojem šalje zahtev, to nije tačno u slučaju ažuriranja. Kada se radi ažuriranje, prvo će se izvršiti transakcija čiji zahtev za 2PC prvi stigne do koordinatora. Koordinator, kao što je već pomenuto u poglavlju ***Koncept rešenja***, upravlja 2PC protokolom, odnosno komunikacijom među centrima za obradu podataka u simuliranom modelu DPSTM sistema.

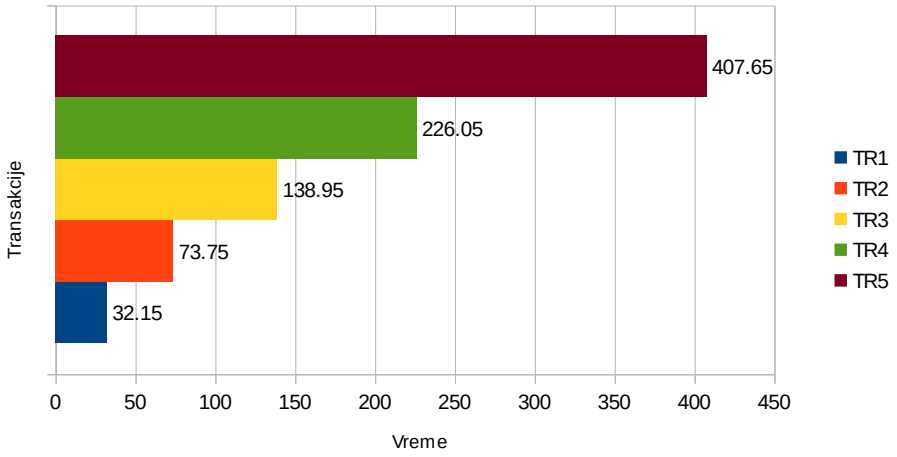
TABELA 7.8

REZULTATI GRUPE TRANSAKCIJA KOJE OBAVLJAJU VALIDACIJU I AŽURIRANJE NA RAZLIČITIM SERVERIMA

ID	Prihvat Zahteva	Početak Ažuriranja	Kraj	Trajanje	Validacija / Ažuriranje	Neuspešno
TR1	5.75	23.35	32.15	8.8	50/10	0
TR2	3.35	64.95	73.75	8.8	100/10	1
TR3	6.95	94.95	138.95	44	10/50	2
TR4	6.05	182.05	226.05	44	100/50	3
TR5	55.65	319.65	407.65	88	10/100	4

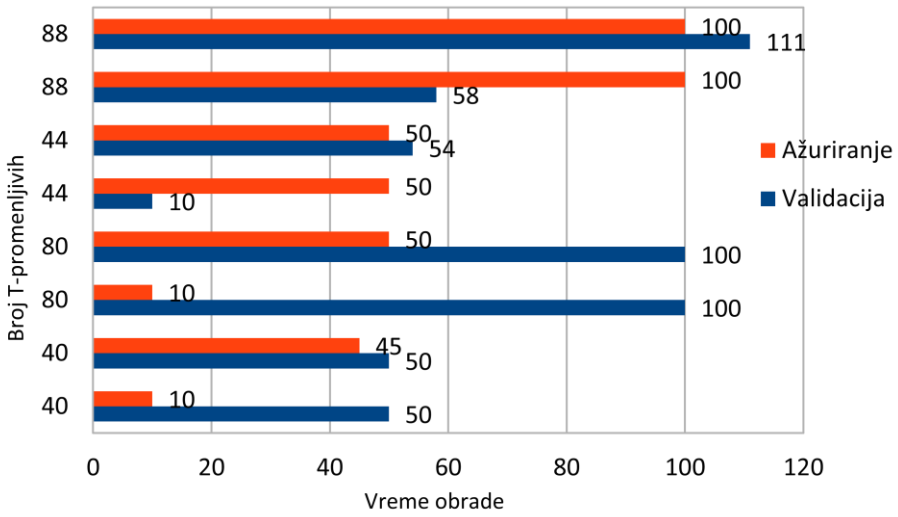
Tabela 7.8 prikazuje rezultate za grupu od pet transakcija poslatih od strane pet različitih korisnika u približno isto vreme. Svaka od transakcija obavlja validaciju nad bar jednom istom i pokušava da ažurira bar jednu istu deljenu t-promenljivu. I u ovom primeru se svaka od transakcija obrađuje na zasebnom centru za obradu podataka. Iz kolone *Prihvat Zahteva* može se primetiti da je zahtev za obradu transakcije TR5 stigao poslednji do servera koji ga obrađuje, iako su svi zahtevi poslati u približno isto vreme. Što dovodi do zaključka da mrežno kašnjenje najviše utiče na redosled obrade transakcija u projektovanom modelu DPSTM sistema.

Razlika u vremenima završetka jedne i početka druge uspešne transakcije nije podjednaka za svaku od transakcija datih u tabeli 7.8. Pomenuta razlika posledica je prvenstveno različitog broja t-promenljivih za koje je potrebno dobaviti najnovije verzije, a potom i razlike u mrežnom kašnjenju za svaku liniju, bilo na relaciji korisnik-centar ili centar-centar.



Slika 7.3Grafička reprezentacija izvršavanja transakcija iz tabele Tabela 7.8 u vremenu

Na slici slici 7.3 grafički su prikazani rezultati izvršavanja transakcija iz tabele 7.8. Kao što se i sa grafika može videti, vreme potrebno da se transakcija uspešno izvrši direktno je proporcionalno broju prethodno neuspešnih izvršavanja, srazmerno broju t-promenljivih nad kojima se tranasakcija obavlja.



Slika 7.4Zavisnost vremena izvršavanja transakcija od broja t-promenljivih i tipa operacija

Grafički prikaz zavisnosti broja t-promenljivih i trajanja izvršenja transakcije prikazani su na slici 7.4. Vreme neophodno da se transakcija obavi ne zavsi samo od broja t-promenljivih nad

kojima ona operira, već i od tipa operacije koju obavlja nad njima. Priroda operacije validacije, jednostavnija je od ažuriranja, te je potrebno manje vremena kako bi se validacija izvršila.

Ukoliko transakcija sadrži skupove t-promenljivih nad kojima se obavlja i validacija i ažuriranje, tada će vreme potrebno da se transakcija izvrši, pod uslovom da nije u konfliktu ni sa jednom drugom, biti jednako većem od: vremena potrebnog da se obavi validacija, odnosno vremena potrebnog da se obavi ažuriranje.

8. Zaključak

U ovom radu razvijen je model prototipa rešenja DPSTM sistema u CloudSim alatu. Predstavljen je način funkcionisanja simulatora i ponašanje sistema. Implementiran je neoptimizovani model prototipa DPSTM sistema. Evaluacija je urađena kroz simulaciju kritičnih slučajeva koji mogu narušiti ispravan rad sistema. Korišćene su sledeće topologije: zvezda, magistrala, nepotpuno i potpuno povezana mreža, i rešetka (engl. *Grid*). Dobijenim rezultatima, koji pokazuju uticaj mreže na redosled, pa samim tim i ishod pojedinih transakcija, verifikovana je ispravnost algoritma obrade transakcija i komunikacije između distribuiranih centara, odnosno komunikacije centara u distribuiranom okruženju.

Kako je koordinator u projektovanom modelu DPSTM sistema unapred poznat, pravac daljeg rada je implementacija i unapređenje simulacije sa izborom lidera (engl. *Leader Election*). Takođe, tu je i razrešavanje mogućih situacija kao što su otkazi servera ili zagušenja mreže. Način na koji bi serveri, u CloudSim alatu, otkrili da li je došlo do otkaza nekog od servera u simuliranom DPSTM sistemu ili je došlo do zagušenja mreže na datoj relaciji ostaje otvoreno pitanje. Na kraju sledi implementacija prototipa i validacija u laboratorijskim uslovima.

9. Literatura

[1] W.Daniel Hillis and Guy L. Steele, Jr: „*Data parallel algorithms“*. Communications of the ACM, 29(12):1170–1183,1986.

[2] David B.Loveman: „*High performance Fortran“*. IEEE Parallel Distrib. Technol. ,1(1):25–42, 1993.

[3] T. Harris, J. R. Larus, i R. Rajwar: „*Transactional Memory“*, 2nd edition, Morgan and Claypool, 2010.

[4] Philip A. Bernstein: „*Transaction processing monitors“*. Communications of the ACM, 33(11):75–86,1990.;

[5] R.Ramakrishnan and J.Gehrke. „*Database Management Systems“*. McGraw-Hill,2000.

[6] Barbara Liskov: „*Distributed programmingin Argus“*. Communications of the ACM, 31(3):300– 312, 1988.

[7] David B. Lomet: „*Process structuring, synchronization, and recovery using atomic actions“*. In ACM Conference on Language Design for Reliable Software, pages 128–137, March 1977. DOI:10.1145/800022.808319 6,62.

[8] Maurice Herlihy and J.Eliot B.Moss: „*Transactional memory: architectural support for-lock free data structures“*. In ISCA ’93: Proc. 20th Annual International Symposium on Computer Architecture,pages 289–300,May 1993.

[9] Janice M.Stone,Harold S.Stone,Phil Heidelberger,and JohnTurek: „*Multiple reservations and the Oklahoma update“*. IEEE Parallel & Distributed Technology, 1(4):58–71, November 1993.

[10] M. Popovic, B. Kordic, I. Bašičević: „*DPM-PSTM: Dual-port Memory Based Python software transactional memory“*, 23nd Telecommunications Forum Telfor (TELFOR), 27-28 Aug. 2015, Belgrade, pp. 1106-1109.

[11] M. Popovic, B. Kordic: „*PSTM: Python software transactional memory“*, 22nd Telecommunications Forum Telfor (TELFOR), 25-27 Nov. 2014, Belgrade, pp. 1106-1109.

[12] H. Quarnoughi, J. Bokhobza, F. Singhoff, S. Rubini: „*Integrating I/Os in Cloudsim for Performance and Energy Estimation“*, ACM SIGOPS Operating Systems Review - Special Topics, 3, December 2016, New York, NY, USA.

[13] Dostupno na: https://en.wikipedia.org/wiki/Data_center, [pristupljeno 25.07.2017].

[14] Dostupno na: https://www.cs.bu.edu/brite/user_manual/node29.html, [pristupljeno 25.07.2017].

[15] B. Wickremasinghe, R. N. Calheiros, R. Buyya, „*CloudAnalyst: A CloudSim-based Visual Modeller for Analysing Cloud Computing Environments and Applications“*, 24th IEEE Internationa Conference on Advanced Information Networking and Applications (AINA). 22-23 April 2010.

[16] A. Kohne, D. Pasternak, L. Nagel, O. Spinczyk: „*Evaluation of SLAbased Decision Strategies for VM Scheduling in Cloud Data Centers“*, In Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud ’16, pages 6:1-6:5, New York, NY, USA, 2016.

[17] Dostupno na:*https://en.wikipedia.org/wiki/Computer_network,* [pristupljeno 25.07.2017].

[18] Groth, David; Toby Skandier, „*Network+ Study Guide, Fourth Edition”*. Sybex, Inc.ISBN 0-7821-4406-3, 2005.

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, i R. Buyya: „*Cloudsim: A toolkit for modeling and simulation of cloud computing enviroments and evaluation of resource provisoning algorithms“*, Software: Practice & Expirience, 41, Jan. 2011.