UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA

Marko Popović

Formalno verifikovana distribuirana softverska transakciona memorija otporna na otkaze

– DOKTORSKA DISERTACIJA –

Mentor:
prof. dr Ilija Bašičević

Novi Sad, 2021

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**:		Докторски рад
Аутор, **АУ**:		Марко Поповић
Ментор, **МН**:		проф. др Илија Башичевић
Наслов рада, НР:		Формално верификована дистрибуирана софтверска трансакциона меморија отпорна на отказе
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Република Србија
Уже географско подручје, **УГП**:		Војводина
Година, **ГО**:		2021.
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)		5 поглавља / 75 страна / 48 цитата / 3 табеле / 8 слика / 9 алгоритама
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Рачунарска техника и рачунарске комуникације
Предметна одредница/Кључне речи, **ПО**:		дистрибуирана трансакциона меморија, формална верификација, отпорност на отказе, детерминизам, репликација података, Пајтон
УДК		
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У дисертацији је развијена дистрибуирана софтверска трансакциона меморија у језику Пајтон, која је формално верификована, отпорна на отказе, детерминистичка, и имплементирана као проширење постојећих апстракција Пајтона. Ово решење је намењено за интелигентне уграђене системе на бази Интернет ствари, тј. за мале и средње ивичне мреже. Приказано решење се састоји од пара трансакционих координатора у режиму водећи-пратећи којим управља дистрибуирани аутомат са коначним бројем стања, и скупа сервера података који се ажурирају детерминистичким протоколом за репликацију података. Формална верификација је урађена конструисањем push/pull семантичког модела и доказивањем одговарајућих критеријума коректности. Експериментални резултати показују супер-линеарно повећање пропусности система, са променом радног оптерећења од оптерећења само са уписима ка оптерећењу само са читањима.
Датум прихватања теме, **ДП**:		30.09.2020.
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	Др Никола Теслић, редовни професор
	Члан:	Др Мило Томашевић, редовни професор
	Члан:	Др Силвиа Гилезан, редовни професор
	Члан:	Др Миодраг Ђукић, доцент
	Члан, ментор:	Др Илија Башичевић, редовни професор

Потпис ментора

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**:	PhD Thesis
Author, **AU**:	Marko Popovic
Mentor, **MN**:	Prof. Ilija Basicevic, PhD
Title, **TI**:	Formally verified fault tolerant 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**:	2021.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovica sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	5 chapters / 75 pages / 48 references / 3 tables / 8 pictures / 9 algorithms
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Computer engineering and communications
Subject/Key words, **S/KW**:	distributed transactional memory, formal verification, fault tolerance, determinism, data replication, Python
UC	
Holding data, **HD**:	The Library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	This dissertation presents a distributed software transactional memory written in Python, which is formally verified, fault tolerant, deterministic, and implemented as an extension of the existing Python abstractions. This solution is primarily targeting intelligent embedded systems based on Internet of things, i.e. small and middle edge networks. The presented solution consists of a pair of transactional coordinators in master-slave mode controlled by a distributed finite state machine, and a set of data servers that are updated by a deterministic data replication protocol. Formal verification is made by constructing a push/pull semantic model and proving the corresponding correctness criteria. Experimental results show superlinear increase of system throughput as a workload changes from the read only to the write only.
Accepted by the Scientific Board on, **ASB**:	30.09.2020.
Defended on, **DE**:	

Defended Board, **DB**:		
	President:	prof. Nikola Teslic, PhD
	Member:	prof. Milo Tomasevic, PhD
	Member:	prof. Silvia Ghilezan, PhD
	Member:	assistant prof. Miodrag Djukic, PhD
	Member, Mentor:	prof. Ilija Basicevic, PhD

Mentor's sign

Iskreno se zahvaljujem svom mentoru prof. Iliji Bašičeviću i prof. Nikoli Tesliću na ukazanom poverenju i podršci u radu, kao i prof. Silvii Gilezan i svima koji su doprineli i omogućili ovaj rad.

Od srca se zahvaljujem dr Branislavu Kordiću i svim svojim kolegama sa kojima sam proveo jedan divan period radeći zajedno i družeći se.

Posebno se zahvaljujem svojoj porodici, bratu Andreju, majci Vlasti i ocu Miroslavu, koji su uvek i bezrezervno bili uz mene pružajući mi oslonac u trenucima kada je to bilo najpotrebnije.

SADRŽAJ

SPISAK ALGORITAMA

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

CSP	Komunicirajući sekvencijalni procesi (eng. Communicating Sequential Processes)
DPSTM	Distribuirana Pajton softverska transakciona memorija (eng. Distributed Python Software Transactional Memory)
DS	Server podataka (eng. Data Server)
DSTM	Distribuirana softverska transakciona memorija (eng. Distributed Software Transactional Memory)
ECM	Režim konačne konzistentnosti (eng. Evenual Consistency Mode)
FSM	Automat sa konačnim brojem stanja (eng. Finite State Machine)
IoT	Internet stvari (eng. Internet of Things)
MSM	Režim vodeći-prateći (eng. Master Slave Mode)
PSTM	Pajton softverska transakciona memorija (eng. Python Software Transactional Memory)
SCM	Režim sekvencijalne konzistentnosti (eng. Sequential Consistency Mode)
STM	Softverska transakciona memorija (eng. Software Transactional Memory)
TA	Vremenski automati (eng. Timed Automata)
TC	Traksakcioni koordinator (eng. Transaction Coordinator)
TM	Transakciona memorija (eng. Transactional Memory)
UML	Ujedinjeni jezik za modeliranje (eng. Unified Modeling Language)

1. UVOD

U ovoj doktorskoj disertaciji je razvijena jedna formalno verifikovana distribuirana softverska transakciona memorija otporna na otkaze. U narednim odeljcima su izloženi predmet, ciljevi, očekivani rezultati (hipoteze), metode i uzorak, i organizacija teksta disertacije.

1.1 Predmet istraživanja

Većina prethodnih istraživanja su rađena na distribuiranim softverskim transakcionim memorijama namenjenim za centre za podatke, i za rezultat su dala rešenja midlvera koja su nedeterministička i nisu za realno vreme, uglavnom pisana u programskim jezicima Java i C++. S druge strane, ugrađeni sistemi zasnovani na Internetu stvari (IoT) u ivičnim mrežama Interneta, kao što su pametne kuće, automobili, itd., treba da rade u realnom vremenu, pa zbog toga treba da budu deterministički. Dodatno, da bi bili inteligentni, ovi sistemi koriste mašinsko učenje, a u današnje vreme Pajton postaje vodeći programski jezik u ovoj oblasti. Konačno, distribuirana softverska transakciona memorija je softverska komponenta kritične infrastrukture i zbog toga ju je potrebno formalno verifikovati, što podrazumeva razvoj odgovarajućeg formalnog modela i dokaz da taj formalni model zadovoljava željena svojstva, kao što je na primer svojstvo mogućnost serijalizacije. U prethodnim istraživanjima su u ovu svrhu korišćeni razni formalni metodi, kao što su vremenski automati i komunicirajući sekvencijalni procesi. Push/Pull semantički model, koji se nedavno pojavio, predstavlja veoma interesantnu alternativu prethodnim metodama, jer

postoji realistično očekivanje da on može biti korišćen generalno za specifikaciju transakciono-zasnovanog konkurentnog softvera.

Predmet (problem) ove doktorske teze je razvoj formalno verifikovane distribuirane softverske transakcione memorije otporne na otkaze, koja je deterministička i implementirana kao prirodno proširenje postojećih apstrakcija programskog jezika Pajton, i čija je formalna verifikacija urađena na osnovu njenog push/pull semantičkog modela.

1.2 Ciljevi istraživanja

Ova doktorska teza ima dva glavna cilja: (1) polazeći od DPSTM v2 treba razviti DPSTM v3 kao sledeći prirodan korak u DPSTM evoluciji, u kom se jedan TC zamenjuje parom TC koji rade u režimu vodeći-prateći (eng. master-slave mode, MSM), kako bi se obezbedila otpornost na otkaze pojedinačnih TC, i (2) treba formalno verifikovati DPSTM v3 na osnovu njenog push/pull semantičkog modela. Konstruisana i verifikovana na ovaj način, DPSTM v3 je jedno rešenje predmeta (problema) ove doktorske teze, tj. jedna formalno verifikovana distribuirana softverska transakciona memorija otporne na otkaze, koja je deterministička i implementirana kao prirodno proširenje postojećih apstrakcija programskog jezika Pajton, i čija je formalna verifikacija urađena na osnovu njenog push/pull semantičkog modela.

Iako je MSM široko studiran i korišćen, u cilju uvođenja para TC koji rade u MSM, potrebno je preispitati ko i kako upravlja parom TC u MSM, zatim prilagoditi DPSTM v2 replikacioni protokol, i na kraju rešiti problem oporavka servisa nakon otkaza koordinatora. S druge strane, da bi se uradila formalna verifikacija DSPTM v3, potrebno je konstruisati odgovarajući push/pull semantički model i dokazati da su zadovoljeni kriterijumi korektnosti.

Iz ovih zadataka, tj. problema koje treba rešiti, proističu glavni očekivani rezultati ove doktorske teze: (1) distribuirani automat sa konačnim brojem stanja za upravljanje parom koordinatora u režimu vodeći-prateći, (2) adaptirtani deterministički protokol za replikaciju podataka, i (3) procedura za oporavak servisa nakon otkaza koordinatora, (4) odgovarajući push/pull semantički model i (5) formalni dokazi da su zadovoljeni kriterijumi korektnosti.

1.3 Očekivani rezultati (hipoteze)

Iz zadataka, tj. problema koje treba rešiti, koji su definisani u odeljku 1.2, proističe i sistem radnih hipoteza ove doktorske teze: (1) moguće je konstruistati distribuirani automat sa konačnim brojem stanja za upravljanje parom koordinatora u režimu vodeći-prateći, (2) moguće je adaptirtati deterministički protokol za replikaciju podataka, (3) moguće je konstruisati proceduru za oporavak servisa nakon otkaza koordinatora, (4) moguće je konstruisati odgovarajući push/pull semantički model i (5) moguće je dokazati da su zadovoljeni kriterijumi korektnosti.

1.4 Metod i uzorak istraživanja

U radu na ovoj doktorskoj tezi su korišćene sledeće metode: (1) analiza i sinteza traženog rešenja u ujedninjenom jeziku za modeliranje (eng. Unified Modeling Language, UML), (2) sinteza komponenata sa pamćenjem u obliku automata sa konačnim brojem stanja (eng. Finite State Machine, FSM), (3) sinteza algoritama u obliku pseudokoda, (4) testiranje putem pojedinačnih testova (eng. unit test) koje pretstavlja oblik automatskog dokazivanja teorema, (5) eksperimentalna evaluacija radi validacije i merenja performanse rešenja, koja se meri propusnošću (broj transakcija u sekundi), za različita radna opterećenja i mrežne konfiguracije, (6) sinteza push/pull semantičkih modela, i (7) dokazivanje teorema o ispunjenosti kriterijuma korektnosti relevantnih push/pull pravila.

U okviru eksperimentalne evaluacije, uzorak je parametrizovana tipična test aplikacija na tipičnim mrežnim konfiguracijama. Test aplikacija sprovodi više faza koje odgovaraju različitim radnim opterećenjima, tj. mešavinama transakcija čitanja i upisa, od mešavine sa 0% čitanja plus 100% upisa, preko niza mešavina sa r% čitanja plus (100 – r)% upisa, do mešavine sa 100% čitanja plus 0% upisa. Dve tipične mrežne konfiguracije su potpuno distribuirana konfiguracija i konfiguracija sa kolokacijom.

U okviru formalne verifikacije, uzorak je generički transakcioni algoritam nad DPSTM v3 u kom transakcija redom: pribavlja t-promenljive, izvodi lokalnu obradu, i aktuelizuje svoja lokalna ažuriranja.

1.5 Organizacija disertacije

Pregled relevantnih radova u oblasti dat je u Poglavlju 2. Rešenje Distribuirane Pajton softverske transakcione memorije otporne na otkaze transakcionih koordinatora (DPSTM v3) je pretstavljeno u Poglavlju 3. Formalna verifikacija DPSTM v3 na osnovu njenog push/pull semantičkog modela je data u Poglavlju 4. Naposletku, originalni doprinosi, mogućnosti primene, prednosti i mane, i pravci budućeg rada su izneti u Poglavlju 5.

2. STANJE U OBLASTI

U prva dva odeljka ovog poglavlja je dato stanje u oblasti kroz pregled relevantnih radova, koji se odnose na distribuirane softverske transakcione memorije i na pristup formalnoj verifikaciji. U trećem odeljku su dati primeri mogućih realnih primena DPSTM v3, a u četvrtom odeljku je opisan pristup eksperimentalnoj evaluaciji koji je korišćen za evaluaciju DPSTM v3.

2.1 Distribuirane softverske transakcione memorije

Herlihi i Mos su prvi uveli mehanizam Transakcione Memorije (TM) [1], kao jedno proširenje protokola koherencije skrivenih (eng. cache) memorija (konkretno u pitanju je Gudmanov "snoopy" protokol za deljenu magistralu [2]). TM podržava transakcije nad objektima u memoriji, a same transakcije se izvršavaju spekulativno, tj. bez zaključavanja, na procesorima sa više jezgara (eng. multicores). Šavit i Touitou su uveli Softversku TM (STM) kao softversku implementaciju TM apstrakcije [3]. Prva Distribuirana STM (DSTM) je uvedena kao proširenje sistema pod nazivom softverska distribuirana deljena memorija (eng. Software Distributed Shared-Memory system, S-DSM), namenjenog za klaster radnih stanica [4]. Slična Java DSTM, zasnovana na detekciji konflikata na nivou objekata i slanju svima (eng. broadcast) informacije o read/write skupovima transakcije u vreme njenog završavanja (eng. commt-time), je predložena u [5-6].

Slično, jedna pouzdana DSTM, nazvana D^{2}STM (od eng. Dependable DSTM) je predložena u [7], kao konačni rezultat intenzivnog razvoja o kom je izveštavano u nizu predhodnih radova [8-11]. D^{2}STM je projektovana na vrhu JVSTM [8], uvođenjem sertifikata Blum flitra (eng. Bloom Filter Certificate, BFC) i procedure za sertifikaciju transakcije u vreme njenog završetka, čime je smanjena sistemska režija po cenu korisnički-podesivog povećanja verovatnoće neuspešnog završetka transakcije (eng. abort).

Veliki varijitet tehnika korišćenih za implementaciju raznih DSTM uključuje sledeće: globalna brava [4], najam serijalizacije, slanje obaveštenja svima u vreme završetka transakcije [7], [12], replikacija i održavanje više verzija t-promenljivih [4], [13], spekulativno izvršenje transakcija u repliciranim okruženjima [14], i raspoređivanje transakcija korišćenjem konzistentnih snimaka [4], [13-14]. Uopšte, razne DSTM koriste jedan od tri moguća modela izvršenja transakcije: (1) model zasnovan na toku upravljanja, u kom su objekti (podaci) nepokretni a transakcije se kreću ili koriste pozive udaljenih procedura [15], (2) model zasnovan na toku podataka, u kom se objekti kreću a transakcije su učvršćene za mrežne čvorove [16-20], i (3) hibridni model u kom su kombinovani modeli sa tokom upravljanja i sa tokom podataka [21-22].

Međutim, sve ove istraživačke inicijative su se uglavnom odnosile na centre za podatke u Internet oblacima, i kao posledica toga, rešenja midlvera koja nude su: (1) nedeterministička, pa time neodgovarajuća za realno vreme, i (2) uglavnom su pisana u programskim jezicima Java i C++.

S druge strane, ugrađeni sistemi zasnovani na IoT u ivičnim mrežama Interenta, kao što su pametne kuće, automobili, itd., treba da rade u realnom vremenu i trebali bi da budu deterministički [23-24]. Takođe, da bi bili inteligentni ovi sistemi koriste mašinsko učenje, i pošto Pajton postaje vodeći programski jezik u ovoj oblasti, ovi sistemi bi trebali da budu zasnovani na Pajtonu. Dakle, pronalaženje rešenja DSTM koje bi istovremeno bilo (1) determinističko i (2) zasnovano na Pajtonu, su dva glavna izazova za ovu doktorsku tezu.

Sledeća linija istraživanja je apostrofirala ove izazove. Python STM (PSTM) je uvedena kao prva STM u Pajtonu. PSTM je formalno verifikovana sa tri komplementarna pristupa, korišćenjem: (1) procesne algebre CSP (od eng.

Communication Sequential Processes) i pratećeg alata PAT (od eng. Process Analysis Toolkit) [26], (2) formalizma vremenskih automata (eng. Timed Automata, TA) i pratećeg alata UPPAAL [27], i (3) Push/Pull semantičkog modela [28]. Arhitektura PSTM raspoređivača, sa četiri onlajn algoritma raspoređivanja transakcija (RR od eng. Round Robin, ETLB od eng. Execution Time Load Balancing, AC od eng. Avoid Conflicts, i AAC od eng. Advanced Avoid Conflicts), je uvdena kao PSTM rukovalac nadmetanjem transakcija (eng. contention manager) u [29], i formalno verifikovana u [30].

Distribuirana PSTM, verzija 1 (DPSTM v1) je uvedena u [31] kao prva DSTM u Pajtonu. DPSTM v1 je formalno verifikovana u [32] generalizacijom push/pull semantičkog modela iz [28]. DPSTM v1 je korisnički-definisan udaljeni menadžer u Pajtonu, koji se izvršava na serveru, dok su transakcije sa svojim zastupnicima njegovi klijenti koji se izvršavaju na udaljenim procesorima, kao što su procesori u IoT uređajima. Očigledno, glavna ograničenja DPSTM v1 su što ona predstavlja: (1) usko grlo za performansu i (2) jednu tačku otkaza celog sistema.

DPSTM v2 je uvedena u [33] kao sledeći prirodan korak u evoluciji DPSTM, u kom je uvedena replikacija podataka [34] radi obezbeđivanja dobre performanse sistema, posebno za transakcije koje rade u režimu konačne konzistentnosti [35]. DPSTM v2 sadrži jednog transakcionog koordinatora (eng. Transaction Coordinator, TC) i n DPSTM v1 replika, nazvanih serverima podataka (eng. Data Server, DS); DS_1 je nazvan *bazna replika*. DPSTM v2 koristi jednostavan i deterministički protokol za replikaciju podataka, skraćeno *replikacioni protokol*, koji podseća na rešenja iz deterističkih baza podataka [36]: po zahtevu transakcije, TC ažurira sve DS, uvek u redosledu od DS_1 do DS_n. Transakcija koja radi u režimu sekvencijalne konzistentnosti (eng. sequential consistency mode, SCM) pribavlja (čita) t-promenljive od TC, dok transakcija koja radi u režimu konačne konzistentnosti (eng. eventual consistency mode, ECM) pribavlja t-promenljive od svog lokalnog DS, koji može biti bilo koji DS_i. Prednosti DPSTM v2 su: (1) dobra performansa za aplikacije sa visokim odnosom broja operacija čitanja prema broju operacija upisa, koje rade u ECM, i (2) $(n - 1)$ otpornost na otkaze (tipa ispada, eng. crashes) DS replika. Očigledno, glavno ograničenje DPSTM v2 je da TC predstavlja jednu tačku otkaza celog sistema.

2.2 Pristup formalnoj verifikaciji

PSTM je najpre formalizovana korišćenjem TA i analizirana pomoću alata za proveru modela UPPAAL [27]. PSTM model zasnovan na TA sadrži automate koji reprezentuju: transakciju, PSTM red čekanja, i transakcionu memoriju (TM). Tri svojstva koja je alat UPPAAL automatski dokazao su: (1) sigurnost (tj. atomičnost), životnost (ovo svojstvo podrazumeva da će se bar jedna od konkurentnih transakcija uspešno završiti – eng. commit), i (3) završetak cikličnih transakcija (ovo svojstvo podrazumeva da će se sve ciklične transakcije nekada konačno završiti).

PSTM je takođe formalizovana korišćenjem CSP i analizirana pomoću alata za proveru modela PAT (eng. Process Analysis Toolkit) [26]. U tom radu su razvijena dva modela: model na višem novou apstrakcije i model na nižem nivou apstrakcije. CSP model na nižem nivou apstrakcije sadrži procese koji reprezentuju: transakciju, API (eng. Application Programming Interface), server, i sistemski rečnik. Tri svojstva koja je alat PAT automatski dokazao su: (1) nepostojanje međusobnog blokiranja (eng. deadlock-freedom), (2) atomičnost-konzistentnost-i-izolacija (eng. Atomicity, Consistency, Isolation – ACI), i (3) optimizam (ovo svojstvo podrazumeva da će se bar jedna od konkurentnih transakcija uspešno završiti).

Tri partikularna algoritma raspoređivanja PSTM transakcija iz [29]: algoritam redom u krug (eng. Round Robin – RR), algoritam uravnoteženja opterećenja na bazi vremena izvršenja (eng. Execution Time Load Balancing – ETLB), i algoritam izbegavanja konflikata (eng. Avoid Conflicts – AC), su formalizovani korišćenjem procesne algebre CSP i analizirani pomoću alata za proveru modela PAT, u cilju njihove evaluacije na osnovu: (1) formalne verifikacije svojstva nepostojanja međusobnog blokiranja i svojstva nepostojanja trajnog blokiranja (ovo svojstvo se još zove svojstvo nepostojanja izgladnjivanja – eng. starvation), i (2) poređenja performanse ova tri algoritma iz perspektive raspona (eng. makespan), ubrzanja (eng. speedup), broja neuspešnih transakcija (eng. aborts), i propusnosti (eng. throughput) [30].

Push/Pull semantički model transakcija se nedavno pojavio kao rešenje koje ujedinjuje širok opseg STM algoritama [37]. Primenom ovog modela moguće je dokazati da zadata STM poseduje tzv. svojstvo *mogućnost serijalizacije* (eng. serializability; sinonim u teoriji distribuiranih sistema je *sekvencijalna konzistentnist*,

eng. sequential consistency), tj. da se transakcije nad njom mogu *serijalizovati* (poređati po nekom serijskom redosledu). Naime, s obzirom da push/pull model zadovoljava svojstvo mogućnost serijalizacije, moguće je dokazati da zadata STM takođe zadovoljava to svojstvo pomoću procedure od sledeća dva koraka: (1) konstruisati push/pull semantički model za zadatu STM, i (2) dokazati da taj model zadovoljava odgovarajuće kriterijume korektnosti. Dokazi kriterijuma korektnosti se tipično oslanjaju na svojstva komutativnosti sekvencijalnih programa [38]. Push/Pull semantički model je veoma privlačan formalizam jer postoji realistično očekivanje da može biti korišćen ne samo za STM već i šire za specifikaciju transakciono-zasnovanog konkurentnog softvera, kao na primer u [39].

Prvi rezultati istraživanja na tragu ove doktorske disertacije su objavljeni u zborniku međunarodne konferencije ECBS 2019 [28]. U tom radu je dokazano da PSTM implementacija zadovoljava svojstvo mogućnost serijalizacije (tj. sekvencijalne konzistentnosti), tako što je konstruisan njen push/pull semantički model i pokazano da taj model zadovoljava kriterijume korektnosti za relevantna push/pull semantička pravila. Istraživanje je zatim prošireno tako da obuhvati formalnu verifikaciju kako lokalne PSTM tako i DPSTM v1, i rezultat je objavljen u međunarodnom časopisu RRST [32]. Konačno, u ovoj doktorskoj disertaciji je istraživanje iz [32] prošireno tako da obuhvati formalnu verifikaciju DPSTM v3.

2.3 Moguće primene

Iako DPSTM v3 primarno cilja inteligentne ugrađene sisteme zasnovane na IoT, kao što su pametne kuće, ona se takođe može koristiti u širokom opsegu aplikacionih domena, od nadzorno-upravljačkih (eng. Supervisory Control and Data Acquisition, SCADA) sistema do simulacija velike skale. Ovde je ukratko dat kratak pregled tri moguće realne primene DPSTM v3.

U rešenju pametne kuće, kao što je jedno predloženo u [40-41], DPSTM v3 se može koristiti kao privremeno skladište (eng. cache) za podatke iz realnog vremena koje prikupljaju konvertori protokola, npr. temperatura, vlažnost, itd., i tako može da zameni moderne komponente koje se koriste u ovu svrhu, kao što je Redis. Glave prednosti DPSTM v3 u odnosu na Redis su što je ona deterministička i što je formalno verifikovana.

U malim i srednjim SCADA sistemima, DPSTM v3 se može koristiti kao transakciona baza podataka u operativnoj memoriji za podatke iz realnog vremena. Dosta je interesantno da dobro poznata OASyS SCADA takođe koristi jedno rešenje baze podataka za podatke iz realnog vremena, koje je kao kod DPSTM v3, zasnovano na korišćenju dve komponente u režimu vodeći-prateći. Međutim, prema saznanjima autora, informacija o tome da li je ovo rešenje i formalno verifikovano nije javno raspoloživa.

U simulacijama velike skale, DPSTM v3 se može koristiti kao skladište za simulacione podatke. Što se toga tiče, PSTM je u ove svrhe već bila korišćena zajedno sa veoma velikim simulacionim programskim paketom pod nazivom DEEPSAM [42], koji je napisan u Pajtonu i Fortranu, i koji se u oblasti računarske-hemije koristi za rešavanje problema predikicije strukture proteina (koja npr. nastaje prilikom interakcije lekova sa ljudskim telom), na serverima sa mnogo jezgara (eng. many-cores). DPSTM v3 bi u budućnosti mogla biti korišćena zajedno sa DEEPSAM za takve simulacije na računarskim klasterima.

2.4 Pristup eksperimentalnoj evaluaciji

Kao što je istaknuto u prethodnom odeljku 2.3, DPSTM v3 može biti korišćena u mnogim realnim aplikacijama. Međutim, u svim tim aplikacijama, ona se uvek koristi kao neka vrsta transakcionog skladišta za podatke, najčešće iz realnog vremema, tako da je njen praktičan zadatak uvek isti: da efikasno podržava transakcije, tj. atomske nizove read i write operacija. Uobičajeno je da se kao mera ove efikasnosti koristi propusnost sistema (tj. broj transakcija ili operacija u jedinici vremena).

Eksperimentalna evaluacija ove vrste sistemskih komponenti je teška, zato što je: (1) broj raspoloživih realnih aplikacija uvek ograničen (u poređenju sa širokim opsegom mogućih aplikacija), i (2) akteri zainteresovani za profit od takvih sistema su obično protiv objavljivanja takvih podataka jer ih smatraju za poslovnu tajnu.

Zbog toga, istraživači kako u akademiji tako i u industriji obično koriste takozvana *sintetička radna opterećenja* da bi okarakterisali (tj. simulirali) širok opseg svih mogućih aplikacija. Tipično, ova radna opterećenja se prave kao različite mešavine read i write operacija. Na primer, eksperimentalna evaluacija dobro poznate komponente Zookeeper, firme Amazon, je urađena korišćenjem ovog metoda, npr. vidi

Sliku 5 u [43], koja pokazuje propusnost sistema (tj. ukupan broj read i write operacija u jedinici vremena) u zavisnosti od učešća (proporcije) operacija čitanja u ukupnom broja operacija u radnom opterećenju. Prateći ovaj faktički standardni metod, test aplikacija konstruisana u ovom istraživanju generiše sintetička radna opterećenja kao mešavine primitivnih read i write transakcija (tj. atomskih read i write operacija), i prezentacija rezultata je analogna sa [43].

Druga poteškoća sa eksperimentalnom evaluacijom ove vrste distribuiranih sistema je što je ona veoma zavisna od korišćenih mrežnih konfiguracija i tehnologija. Kao što postoji širok opseg mogućih aplikacija, tako isto postoji širok opseg mogućih mrežnih infrastruktura, koji je teško pokriti.

U preliminarnoj eksperimentalnoj evaluaciji koja je urađena u ovom istraživanju, korišćena je mrežna tehnologija koja je bila raspoloživa u laboratoriji (tipična moderna LAN mreža), i dve mrežne konfiguracije (potpuno distribuirana konfiguracija i konfiguracija sa kolokacijom) koje se sa stanovišta DPSTM v3 mogu posmatrati kao dva suprotna ekstrema u celom opsegu mogućih mrežnih konfiguracija. Planirano je da se u budućem radu izvedu dodatni eksperimenti na drugim mrežnim konfiguracijama i tehnologijama.

3. DPSTM v3

U ovom poglavlju je predstavljena DPSTM v3, koja je realizovana da ispuni prvi cilj ove doktorske disertacije, a on je da se polazeći od DPSTM v2, evolutivno razvije rešenje otporno na otkaze transakcionih koordinatora. U naredna tri odeljka (čija preliminarna verzija je izašla u [44]) su redom izloženi: DPSTM v3 projekat, DPSTM v3 implementacija, i eksperimentalna evaluacija DPSTM v3.

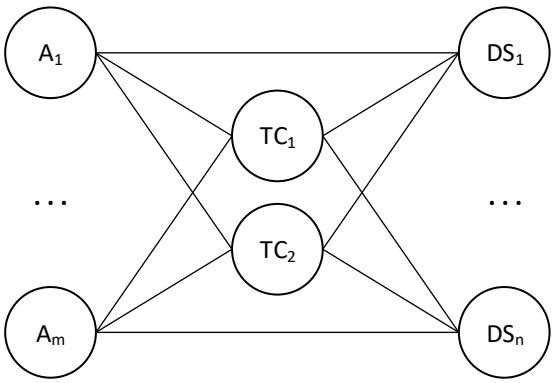
3.1 DPSTM v3 projekat

Ovaj odeljak predstavlja DPSTM v3 projekat na visokom nivou apstrakcije. Sledeća tri pododeljka predstavljaju arhitekturu sistema zasnovanog na DPSTM v3, DPSTM v3 interfejs za programiranje aplikacija (API), i rad DPSTM v3, tj. njeno ponašanje.

3.1.1 Arhitektura sistema zasnovanog na DPSTM v3

Sistem zasnovan na DPSTM v3 je klijent-server arhitektura sa više servera u dva nivoa, koja sadrži *m* klijentskih aplikacionih programa, A_1, …, A_m, i DPSTM v3, koja se dalje sastoji od para TC koji rade u MSM režimu, TC_1, i TC_2, i *n* servera podataka, DS_1, …, DS_n, vidi Sliku 1. U toku normalnog rada sistema, jedan TC je vodeći (eng. master TC, MTC), dok je drugi prateći (eng. slave TC, STC). MTC i STC rade kao par aktivan-pripravan, što znači da je MTC aktivan, a STC samo čeka da preuzme sistem ako i kada MTC otkaže. U slučaju da MTC otkaže, i tokom njegove popravke, sistem nastavlja da radi sa jednom TC, koji radi kao solo MTC.

Svaki aplikacioni program A_i može da sadrži više transakcija koje rade u SCM ili ECM. Transakcija koja radi u SCM zahteva servise samo od MTC, a on dalje zahteva servise od prisutnih DS, i u ovom slučaju se arhitektura sistema može posmatrati kao troslojna arhitektura. Ako pak transakcija radi u ECM, ona zahteva usluge od MTC i od svog lokalnog DS koji se nalazi u istom računaru ili u nekom računaru u blizini, i u ovom slučaju se arhitektura sistema može posmatrati kao troslojna arhitektura sa dodatnom direktnom vezom između prvog i trećeg sloja.



Slika 1 – Arhitektura sistema zasnovanog na DPSTM v3

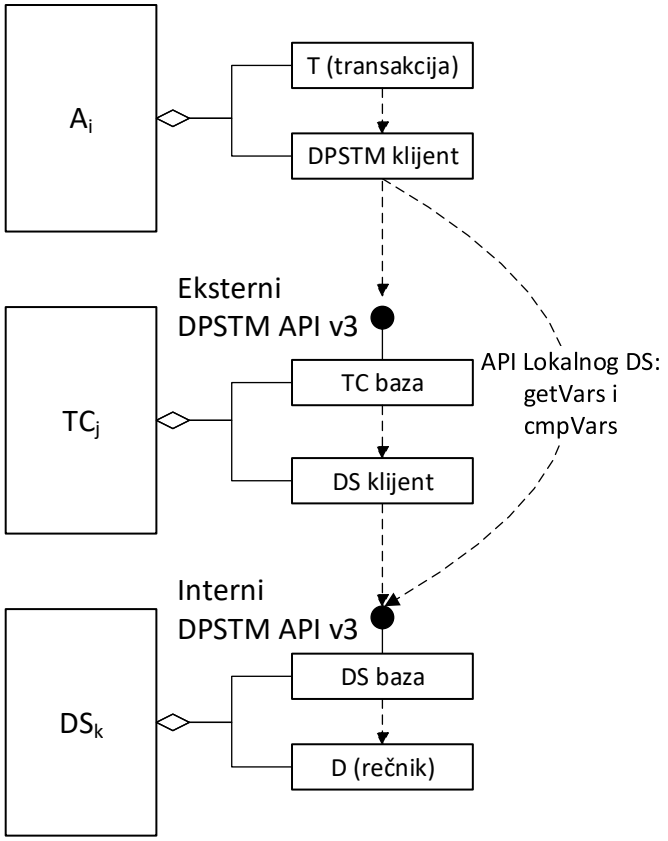
Serveri (TC i DS) su projektovani da budu menadžeri iz Pajton (ver. 3) modula "multiprocessing" sa namerom da se arhitektura sa Slike 1 može implementirati kao prirodno proširenje postojećih apstrakcija programskog jezika Pajton.

Slika 2 prikazuje pojednostavljen UML dijagram klasa sistema zasnovanog na DPSTM v3. Uopšte, svaki aplikacioni program A_i sadrži jednu ili više transakcija (T) i pratećih DPSTM klijenata koji rade kao zastupnici transakcija, svaki transakcioni koordinator TC_j sadrži TC bazu i DS klijenta koji radi kao zastupnik TC_j, i svaki server podataka DS_k sadrži DS bazu i rečnik D koji sadrži sve t-promenljive.

DPSTM klijent može biti konfigurisan za SCM ili ECM, dok je DS klijent uvek konfigurisan za SCM. TC nudi eksterni DPSTM API v3 svim DPSTM klijentima, dok DS nudi potpun interni DPSTM API v3 svim DS klijentima i lokalni DS API svim DPSTM klijentima koji su konfigurisani za ECM.

U toku pokretanja sistema, klijenti uspostavljaju veze zaštićene autentifikacijom sa svojim serverima. Preciznije govoreći, DS klijent uspostavlja veze sa svim DS, DPSTM klijent konfigurisan za SCM uspostavlja veze sa oba TC, i DPSTM klijent konfigurisan za ECM uspostavlja veze sa oba TC i sa svojim lokalnim DS.

U toku normalnog rada sistema, DPSTM klijent konfigurisan za ECM jednostavno delegira sve zahteve transakcija vodećem TC. S druge strane, DPSTM klijent konfigurisan za ECM delegira zahteve transakcija za čitanje i poređenje t-promenljivih (vidi funkcije getVars i cmpVars u Odeljku 3.1.2) svom lokalnom DS, a sve druge zahteve transakcija vodećem TC.



Slika 2 – UML klas dijagram sistema zasnovanog na DPSTM v3

3.1.2 DPSTM v3 API

Kao što je spomenuto u Odeljku 3.1.1 i prikazano na Slici 2, unutar DPSTM v3 postoje tri različita API: (1) eksterni DPSTM API v3, (2) interni DPSTM API v3, i (3) API lokalnog DS.

Eksterni DPSTM API v3 je napravljen od DPSTM API v2 (vidi [27]) uvođenjem novog argumenta *txnid* na kraju liste argumenata svake API funkcije (*txnid* je identifikacija transakcije; ove identifikacije transakcija se koriste interno kako bi se API pozivi izdati od različitih transakcija mogli razlikovati). Međutim, DPSTM API v2 je zadržan kao interni interfejs između transakcije T i DPSTM klijenta, unutar A_i

(vidi Sliku 2), kako bi se omogućilo jednostavno prenošenje postojećeg softvera na DPSTM v3.

Četiri najvažnije funkcije eksternog DPSTM v3 su: (1) addVars koja deklariše (dodaje) nove t-promenljive, (2) putVars koja postavlja početne vrednosti zadatih t-promenljivih, (3) getVars koja vraća (čita) vrednostih zadatih t-promenljivih, koje se dalje koriste kao lokalne t-promenljive unutar funkcije za lokalnu obradu koju izvršava transakcija, i (4) commitVars koja proverava da li su t-promenljive nad kojim je transakcija operisala aktuelne, i ako jesu, izvršava ažuriranja (upise) učinjena nad lokalnim t-promenljivama na globalne (deljene) t-promenljive koje se drže u rečniku unutar DS.

Interni DPSTM API v3 proširuje eksterni DPSTM API v3 sa sledeće dve funkcije: (1) getLastOp vraća torku (*opc*, *opa*, *opr*, *txnid*) gde su *opc*, *opa*, *opr*, i *txnid* redom: kod poslednje izvršene operacije (tj. ime API funkcije), njeni argumenti, njena povratna vrednost, i identifikacija (ID) transakcije, i (2) execLastOp izvršava operaciju zadatu putem torke (*opc*, *opa*, *opr*, *txnid*), tj. izvršava *opc* sa argumentima *opa*.

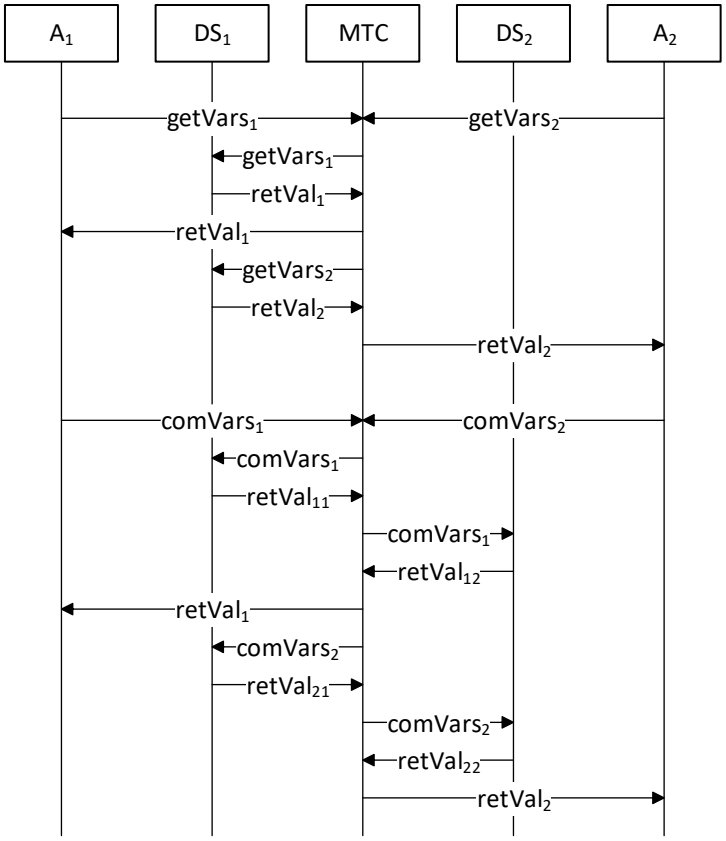
API lokalnog DS je interni DPSTM API v3 redukovan na sledeće dve funkcije: (1) getVars (vidi gore), i (2) cmpVars koja poredi lokalne t-promenljive sa odgovarajućim globalnim t-promenljivama koje se drže u rečniku u DS.

3.1.3 DPSTM v3 ponašanje

U ovom odeljku je dat pregled ponašanja DPSTM v3 kroz predstavljanje sledeća tri najvažnija scenarija: (1) par aplikacionih procesa koji rade u SCM, (2) par aplikacionih procesa koji rade u režimu ECM, i (3) jedan primer prevazilaženja otkaza MTC. Na slikama u ovom odeljku (Slike 3 do 5), ime API funkcije commitVars je skraćeno na comVars kako bi slike bile čitljive. Radi jednostavnosti, predpostavlja se da je $n = 2$, tj. da u sistemu postoje dva servera podataka (DS_1 i DS_2).

Slika 3 prikazuje tipičan scenario sa parom aplikacionih procesa, A_1 i A_2, koji rade u SCM. Na početku, A_1 i A_2 simultano izdaju svoje getVars pozive da bi pribavili svoje kopije t-promenljivih – ovi pozivi su na Slici 3 prikazani redom kao $getVars_1$ i $getVars_2$. Pretpostavimo da je MTC besposlen i da $getVars_1$ stiže do MTC pre $getVars_2$, tako da se $getVars_1$ ulačava u red zahteva ka MTC pre $getVars_2$. Nakon što $getVars_1$ stigne na početak read zahteva ka MTC, MTC poslužuje $getVars_1$ čitanjem t-

promenljivih koje zahteva A_1 i vraćanjem njihovih kopija ka A_1 preko povratne vrednosti $retVal_1$. Zatim, MTC poslužuje $getVars_2$ čitanjem t-promenljivih koje zahteva A_2 i vraćanjem njihovih kopija ka A_2 preko povratne vrednosti $retVal_2$. Primetite kako MTC serijalizuje $getVars_1$ i $getVars_2$ kako bi im obezbedio aktuelne (tj. poslednje) verzije traženih t-promenljivih.

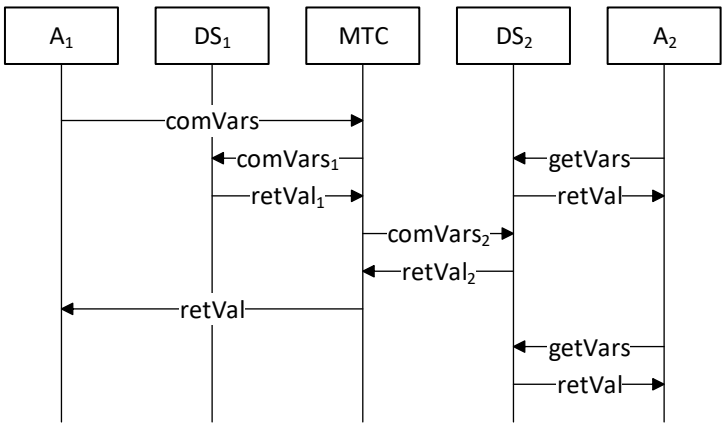


Slika 3 – Par aplikacionih procesa koji rade u SCM

Na kraju, A_1 i A_2 simultano izdaju svoje commitVars pozive da bi se aktuelizovala njihova ažuriranja na globalnim t-promenljivama – ovi pozivi su na Slici 3 prikazani redom kao $comVars_1$ i $comVars_2$. Pretpostavimo da se $comVars_1$ ulančava u red zahteva ka MTC pre $comVars_2$. Nakon što $comVars_1$ stigne na početak reda zahteva ka MTC, MTC poslužuje $comVars_1$ delegiranjem ovog zahteva prvo ka DS_1. Po prijemu povratne vrednosti $retVal_{11}$, MTC proverava da li je $comVars_1$ bio uspešan (tj. da li $retVal_{11}$ sadrži kod 'yes'), i ako jeste (kao što je pretpostavljeno na Slici 3), onda MTC delegira isti zahtev ka DS_2. Posle prijema povratne vrednosti $retVal_{12}$, MTC je proverava i vraća konačnu povratnu vrednost $retVal_1$ ka A_1. U slučaju da

$comVars_1$ koji je bio delegiran DS_1 nije bio uspešan (što nije slučaj na Slici 3), MTC odmah vraća odgovarajuću povratnu vrednost $retVal_1$ ka A_1.

Posle posluživanja $comVars_1$, MTC poslužuje $comVars_2$ analogno kao što je posluživao $comVars_1$, i na kraju, šalje $retVal_2$ ka A_2. Primetite kako MTC ponovo serijalizuje $comVars_1$ i $comVars_2$ kako bi obezbedio atomičnost odgovarajućih ažuriranja. Takođe, primetite kako MTC delegira $comVars_1$ (i $comVars_2$) ka DS_1 i ka DS_2, kako bi obezbedio (tj. sproveo) replikaciju podataka.



Slika 4 – Par aplikacionih procesa koji rade u ECM

Slika 4 prikazuje tipičan scenario sa parom aplikacionih procesa, A_1 i A_2, koji rade u ECM, gde je A_1 proces tipa proizvođača, koji periodično ažurira neke t-promenljive, dok je A_2 proces tipa potrošača, koji periodično čita neke t-promenljive (koristeći funkciju getVars) iz svog lokalnog servera podataka DS_2 i zatim radi neku obradu podataka nad tim očitanim vrednostima, koje ne moraju da budu sveže (tj. poslednje). Radi jednostavnosti, Slika 4, prikazuje samo jednog proizvođača i jednog potrošača, ali u opštem slučaju u sistemu može biti više proizvođača i potrošača. Takođe radi jednostavnosti, Slika 4 prikazuje samo najvažniji deo rada sistema, koji se fokusira na commitVars pozive od A_1 i getVars pozive od A_2, pri čemu se ovi pozivi poslužuju paralelno od strane redom MTC i DS_2.

Na početku, A_1 izdaje svoj commitVars poziv ka MTC (prikazan kao comVars na Slici 4), a MTC sa svoje strane koristi već opisan deterministički replikacioni protokol da bi ažurirao DS_1 i DS_2 (u tom redosledu). MTC sprovodi ovaj protokol sekvencijalno delegirajući comVars ka DS_1 i DS_2 (koji su redom prikazani kao $comVars_1$ i $comVars_2$ na Slici 4). Primetite da nakon što je bazna replika DS_1

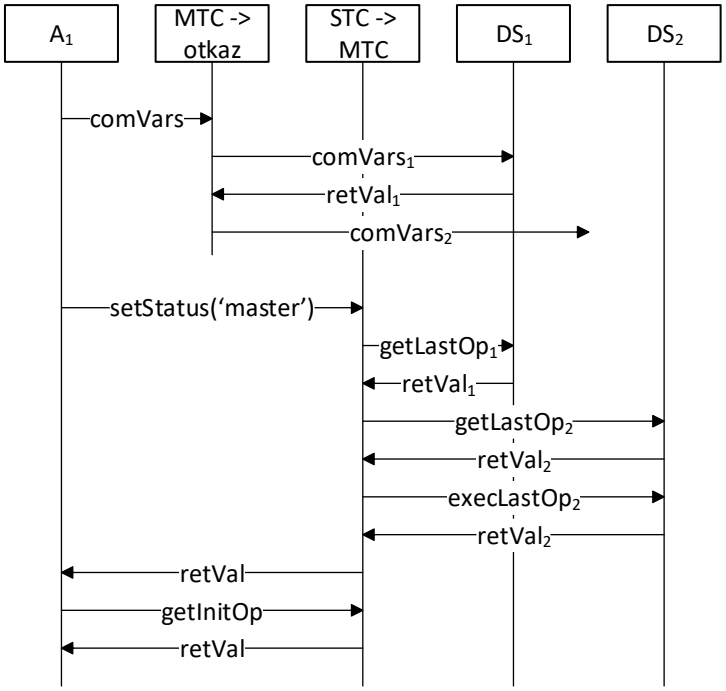
ažurirana, sistem ulazi u nekonzistentno stanje, zato što neke t-promenljive u DS_1 i DS_2 imaju različite verzije i vrednosti. Međutim, sistem se vraća nazad u konzistentno stanje odmah nakon što je DS_2 takođe ažuriran – ovo svojstvo sistema se zove konačna konzistentnost. Takođe primetite da dok je sistem u nekonzistentnom stanju, postoje najviše dve verzije neke t-promenjive koja prolazi kroz ažiriranje – njena tekuća vrednost u baznoj replici (i u drugim DS koji su već ažurirani) i njena prethodna verzija u DS koji još nisu ažurirani (na Slici 4 ovo može biti samo DS_2).

Stoga, aplikacioni proces koji koristi svoj sopstveni lokalni DS (kao što na Slici 4, A_2 koristi DS_2) može da pribavi ili prethodnu verziju ili tekuću verziju tražene t-promenljive. Slika 4 ilustruje oba slučaja. Prvi getVars poziv od A_2 se poslužuje od strane DS_2 za vreme dok DS_1 poslužuje $comVars_1$, i za vreme dok DS_2 još nije bio ažuriran, tako da odgovarajuća retVal vraćen ka A_2 sadrži prethodnu verziju (i vrednost) zahtevane t-promenljive. S druge starne, drugi getVars poziv od A_2 se poslužuje od strane DS_2 nakon što je DS_2 bio ažuriran, tako da odgovarajuća retVal vraćena ka A_2 sadrži tekuću verziju (i vrednost) zahtevane t-promenljive.

Očigledno, korišćenje lokalnih servera podataka dovodi do bolje performanse (po cenu mogućeg korišćenja prethodnih verzija t-promenljivih). Primetite da konfigurisanje sistema u kom aplikacioni procesi mogu (statički ili dinamički) da biraju svoje lokalne servere iz skupa (geografski) bliskih DS, otvara mogućnost za balansiranje getVars poziva. Za sada, projektant sistema zasnovanog na DPSTM v3 mora ručno da obavi statičko uravnoteženje opterećenja putem specifikacije lokalnih DS u konfiguracionim datotekama. Razvoj rešenja za automatsko balansiranje opterećenja, možda korišćenjem nekih tehnika iz [45] je jedna od mogućih tema za buduća istraživanja.

Slika 5 prikazuje tipičan scenario oporavka servisa nakon ispada MTC, koji je iniciran od aplikacionog procesa A_1 nakon što je regularno posluživanje njegovog commitVars poziva (ili bilo kog drugog API poziva sa ivičnim efektima) bilo prekinuto zbog otkaza MTC (koji se desio tokom regularnog replikacionog protokola i u nekoj tački nakon što je bazna replika bila uspešno ažurirana). Pored ovog konkretnog scenaria, postoje još tri prilično slična scenarija, koji će biti pokriveni kasnije u Odeljku 3.2 (DPSTM v3 implementacija).

U scenariju na Slici 5, DPSTM klijent transakcije, unutar A_1, na početku interno startuje vremensku kontrolu i izdaje svoj commitVars poziv ka MTC, koji sa svoje strane započinje izvršavanje redovnog protokola za replikaciju podataka i uspešno ažurira baznu repliku DS_1 (vidi $comVars_1$ i odgovarajuću $retVal_1$ na Slici 5). Međutim, pre izdavanja $comVars_2$, MTC otkazuje, i zbog toga on nikada ne šalje retVal ka DPSTM klijentu transakcije. Primetite da je u ovoj tački sistem u nekonzistentnom stanju zato što serveri podataka sadrže različite podatke.



Slika 5 – Tipičan scenario prevazilaženja otkaza MTC

Nakon isteka zadatog vremenskog intervala, aktivira se interna vremenska kontrola, čime DPSTM klijent transakcije detektuje otkaz MTC i inicira prevazilaženje tog otkaza (tj. prekopčavanje STC u novi MTC) izdavanjem setStatus poziva ka STC, sa argumentom 'master'. U ovoj tački, STC se prekopčava u MTC i započinje proceduru oporavka servisa nakon otkaza koorinatora (koji se desio u toku izvršenja protokola za replikaciju podataka) kako bi vratio sistem u konzistentno stanje (u kom svi serveri podataka sadrže iste podatke).

Novi MTC oporavlja servis commitVars poziva nakon otkaza koordinatora na sledeći način. U prvom koraku, MTC izdaje getLastOp poziv ka baznoj replici DS_1 (prikazan kao $getLastOp_1$ poziv na Slici 5) i pribavlja podatke o poslednjoj operaciji op_1 (tj. o servisu poslednjeg API poziva) koju je obavio DS_1; te podatke dobija unutar

povratne vrednosti $retVal_1$. U drugom koraku, MTC izdaje getLastOp poziv ka DS_2 (prikazan kao $getLastOp_2$ poziv na Slici 5), pribavlja podatke o poslednjoj operaciji op_2 koju je obavio DS_2, poredi podatke o op_2 i op_1 i otkriva da su oni različiti, tj. da je sistem u nekonzistentnom stanju. U trećem koraku, MTC zahteva od DS_2 da obavi operaciju op_1 (koju je obavila bazna replika) izdavanjem execLastOp poziva ka DS_2 (prikazan kao $execLastOp_2$ na Slici 5), pribavlja povratnu vrednost te operacije unutar povratne vrednosti $retVal_2$ od DS_2, i utvrđuje da je ona ista kao povratna vrednost za op_1. Primetite da se u opštem slučaju sa n servera podataka, koraci dva i tri ponavljaju $(n - 1)$ puta. Primetite takođe da kada se ova procedura završi, sistem se vraća u konzistentno stanje.

Nakon što novi MTC oporavi prekinuti commitVars poziv, on vraća povratnu vrednost retVal DPSTM klijentu transakcije unutar A_1, koji sa svoje strane izdaje getInitOp poziv prema novom MTC kako bi pribavio podatke o oporavljenoj operaciji, tj. op_1, i vraća povratnu vrednost od op_1 ka transakciji unutar A_1. Sistem dalje nastavlja da radi sa jednim MTC sve dok otkazali TC ne bude popravljen i ponovo uključen u rad.

3.2 DPSM v3 implementacija

Ovaj odeljak prikazuje klučne elemente DPSTM v3 implementacije, koji predstavljaju glavne doprinose ovog rada. Sledeća četiri pododeljka predstavlaju: (1) distribuirani automat sa konačnim brojem stanja za režim vodeći-prateći (MS-FSM), (2) distribuirano prevazilaženje otkaza koordinatora, (3) adaptirani deterministički protokol za replikaciju podataka, i (4) proceduru za oporavak servisa nakon otkaza koordinatora.

3.2.1 Distribuirana MS-FSM

Rešenje distribuirane MS-FSM podrazumeva da je korišćena komunikaciona infrastruktura pouzdana, tj. u terminima CAP teoreme [46], ono žrtvuje otpornost na particionisanje mreže za konzistentnost i raspoloživost.

Distribuirana MS-FSM je implementirana kao skup MS-FSM instanci, jedna instanca po transakciji, gde je svaka instanca MS-FSM sadržana unutar objekta odgovarajućeg DPSTM klijenta. Pojedinačna instanca MS-FSM je implementirana kao par objekata zastupnika TC i dve funkcije DPSTM klijenta, nazvane getTCs i doOp,

koje izvode prelaze između stanja MS-FSM. Ove dve funckije su hijerarhijski organizovane, tako da getTCs izvodi prelaze zasnovane ne tekućem stanju raportiranom od para TC, dok doOp izvodi isključivo prelaze koji se tiču prevazilaženja otkaza koordinatora, i ona koristi getTCs kada je to potrebno. Stoga je funkcija getTCs podređena funkciji doOp. Radi jasnoće, ovaj odeljak opisuje getTCs dok Odeljak 3.2.2 (distribuirano prevazilaženje otkaza) opisuje doOp.

Funkcija getTCs konstruiše i vraća par objekata zastupnika TC (*tcm*, *tcs*), gde su *tcm* i *tcs* redom zastupnici od MTC i STC. Objekti *tcm* i *tcs* se skladište u odgovarajućim poljima objekta DPSTM klijenta.

Tabela 1 – MS-FSM prelazi koje izvodi getTCs

Tekuće stanje		Naredno stanje		Izlaz (*tcm*, *tcs*)	
TC_1	TC_2	TC_1	TC_2	*tcm*	*tcs*
isključen	isključen	isključen	isključen	nema	nema
prateći	isključen	vodeći	isključen	tc_1	nema
isključen	prateći	isključen	vodeći	tc_2	nema
prateći	prateći	vodeći	prateći	tc_1	tc_2
vodeći	isključen	vodeći	isključen	tc_1	nema
isključen	vodeći	isključen	vodeći	tc_2	nema
vodeći	prateći	vodeći	prateći	tc_1	tc_2
prateći	vodeći	prateći	vodeći	tc_2	tc_1
vodeći	vodeći	isključen	isključen	Greška	Greška
Napomena: TC prelaz iz “vodeći” u “prateći” je zabranjen.					

Inicijalno, transakcija poziva konstruktor DPSTM klijenta i prosleđuje mu identifikaciju transakcije (*txnid*) kao argument, a konstruktor sa svoje strane poziva funkciju getTCs radi popunjavanja polja zastupnika TC. Funkcija getTCs zatim efektivno konstruiše par objekata zastupnika (*tcm*, *tcs*) i ažurira stanja dva TC u sistemu (interno predstavljenih pomoću objekata tc_1 i tc_2) izvršavajući zahtevani prelaz između stanja u skladu sa Tabelom 1.

Pojedinačan TC može biti u jednom od sledeća tri moguća stanja: “isključen” (tj. nije u radu), “prateći”, i “vodeći”. Zbog toga, par TC, (TC_1, TC_2), teorijski može biti u jednom od devet mogućih stanja (3 x 3 = 9), vidi Tabelu 1. Deveto stanje (“vodeći”, “vodeći”) je zabranjeno (ulazak u ovo stanje odgovara fatalnoj sistemskoj grešci). Slično, prvo stanje (“isključen”, “isključen”) takođe nije validno radno stanje, jer sistem ne može da radi bez TC.

Po projektu, dozvoljeni prelazi između stanja za pojedinačni TC su: (1) iz "isključen" u "isključen" ili "prateći", (2) iz "prateći" u "prateći" ili "vodeći", i (3) iz "vodeći" u "vodeći". Prelaz između stanja br. 1 izvodi TC samostalno, dok se prelazi između stanja br. 2 i 3 izvode unutar getTCs putem setStatus poziva nad objektom zastupnika TC. Takođe po projektu, stanje pojedinačnog TC se ne može promeniti iz "vodeći" u "prateći".

Primetite da neki prelazi između stanja u Tabeli 1 ne menaju stanja para TC. Preciznije, stanje se menja samo za sledeća tri tekuća stanja: ("prateći", "isključen"), ("isključen", "prateći"), i ("prateći", "prateći"); za druga validna tekuća stanja, sledeće stanje je isto kao tekuće stanje. Primetite takođe da ako više DPSTM klijenata izda isti setStatus poziv paralelno, prvi posluženi poziv će biti uspešan dok su preostali pozivi bezopasni i bez dejstva.

3.2.2 Distribuirano prevazilaženje otkaza

Primarni zadatak funkcije doOp je da servisira API pozive izdate od strane transakcija. Po projektu, servisiranje API poziva se naziva operacijom, *Op* = (*code*, *args*, *ret*, *txnid*), gde su *code*, *args*, *ret*, i *txnid* redom: ime API funkcije (npr. addVars, putVars, itd.), argumenti API funkcije, povratna vrednost API funkcije, i identifikacija transakcije. Radi referenciranja elemenata neke instance *op* od *Op*, koristiće se uobičajena notacija sa tačkom, npr. *op.code* je element *code* od *op*, itd.

Tabela 2 – MS-FSM prelazi koje izvodi doOP

Tekuće stanje		Naredno stanje		Izlaz (*tcm*, *tcs*)	
TC_1	TC_2	TC_1	TC_2	*tcm*	*tcs*
vodeći	prateći	isključen	vodeći	tc_2	nema
prateći	vodeći	vodeći	isključen	tc_1	nema

Sekundarni zadatak funkcije doOp je da detektuje otkaz MTC i inicira prevazilaženje tog otkaza izvršavanjem zahtevanog prelaza između stanja u skladu sa Tabelom 2.

Pseudokod funkcije doOp je dat u Algoritmu 1. Postoji pet mogućih putanja kroz funkciju doOp, nazvanih: putanja 1, putanja 2.1, putanja 2.2, putanja 3.1, i putanja 3.2. Putanje 2.1 i 2.2 su podputanje od putanje 2, koja se zove *redovno prevazilaženje otkaza*. Slično, putanje 3.1 i 3.2 su podputanje od putanje 3, koja se zove *brzo prevazilaženje otkaza*.

Algoritam 1 – Pseudokod funkcije doOp

```
01: doOp(op)
02:   try
03:     delegate op to tcm  // putanja 1: normalan rad
04:   exception
05:     if tcs != None
06:       try
07:         // putanja 3: brzo prevazilaženje otkaza
08:         tcs.setStatus('master')
09:         initOp = tcs.getInitOp()
10:         tcm := tcs, tcs := None
11:         if op = initOp
12:           return initOp.ret     // putanja 3.1
13:         else
14:           delegate op to tcm  // putanja 3.2
15:       exception // pada na putanju 2
16:     // putanja 2: redovno prevazilaženje otkaza
17:     tcm, tcs = getTCs()
18:     initOp = tcm.getInitOp()
19:     if op = initOp
20:       return initOp.ret     // putanja 2.1
21:     else
22:       delegate op to tcm  // putanja 2.2
```

Primetite da više DPSTM klijenata može izvršavati svoje kopije doOp funkcije paralelno. Takođe, za jednu od njih, njena operacija (tj. servis) je narušena zbog otkaza MTC. Neka su c_1 i c_2 dve Bulove promenljive pridružene nekom DPSTM klijentu *c*, takve da je c_1 tačno ako je operacija od klijenta *c* narušena, dok je c_2 tačno ako objekat *tcs* od klijenta *c* nije None (tj. *tcs* je povezan sa STC).

Očigledno, postoje četiri moguća slučaja za nekog klijenta *c* koji izvršava doOp: (1) c_1 je tačno i c_2 je tačno (putanja 3.1), (2) c_1 je netačno i c_2 je tačno (putanja 3.2), (3) c_1 je tačno i c_2 je netačno (putanja 2.1), i (4) c_1 je netačno i c_2 je netačno (putanja 2.2).

Razjasnimo kako se dešavaju prelazi između stanja u Tabeli II. Bez gubitka opštosti, predpostavimo da je tekuće stanje od MS-FSM za *c* (“vodeći”, “prateći”) i da se zbog otkaza MTC realno stanje sistema promenilo u (“isključen”, “prateći”). Ako je c_2 tačno i STC je u radu, *c* prati putanju 3 i direktno menja stanje svoje MS-FSM u (“isključen”, “vodeći”) izvršenjem linije 10 u Algoritmu 1. Analogna analiza se može sprovesti za slučaj kada je tekuće stanje od MS-FSM za *c* (“prateći”, “vodeći”).

Primetite da ako je c_2 netačno, *c* prati putanju 2, na kojoj poziva getTCs, i unutar getTCs izvodi prelaze između stanja u skladu sa Tabelom 1.

3.2.3 Prilagođeni deterministički replikacioni protokol

MTC koristi prilagođeni deteterministički replikacioni protokol kada poslužuje API pozive sa ivičnim efektima, koji dovode do operacija upisa unutar DS (addVars, putVars, commitVars, itd.). Jezgro ovog protokola je dosta jednostavno, i praktično isto kao originalni protokol korišćen u DPSTM v2, po kom MTC sekvencijalno delegira API poziv, koji treba poslužiti, redom svakom DS, i uvek u redosledu od DS_1 (bazna replika) do DS_n. Podsetite se kako MTC poslužuje dolazne commitVars pozive u tipičnom scenariju prikazanom na Slici 3.

Razlika između adaptiranog i originalnog protokola je u načinu kako DS serveri poslužuju API pozive. U originalnom protokolu DS samo izvodi zahtevane operacije na svom rečniku, dok u adaptiranom protokolu DS takođe čuva ovu operaciju kao poslednju operaciju koju je izveo. DS čuva samo jednu jedinu poslednju operaciju, a ne istoriju (log) svih svojih operacije.

3.2.4 Procedura za oporavak servisa nakon otkaza

Novi MTC (tj. bivši STC) koristi proceduru za oporavak servisa API poziva (tj. operacije) kako bi vratio sistem iz nekonzistentnog stanja nazad u konzistentno stanje, kao što je već objašnjeno u tekstu i ilustrovano na Slici 5 u Odeljku 3.1.3 (DPSM v3 ponašanje).

Jezgro ove procedure je dosta jednostavno. Na početku, novi MTC pribavlja poslednju operaciju koju je izveo server podataka DS_1 (bazna replika) pozivanjem funkcije getLastOp nad zastupnikom DS_1, i skladišti je u lokalnoj promenljivoj *initOp* (za novi MTC ovo je početna operacija od koje on kreće). Nakon toga, STC izvodi jednu petlju kako bi proverio i po potrebi oporavio preostale DS. Oporavak nekog DS znači njegovo poravnavanje sa DS_1, jer je DS_1 bazna replika. Unutar ove petlje, novi MTC pribavlja poslednju operaciju koju je izveo DS_i, ($i = 2, \dots, n$), *lastOp*, i poredi je sa svojom *initOp*. Ako *lastOp* nije jednaka sa *initOp*, novi MTC zahteva od DS_i da izvede *initOp* pozivanjem funkcije execLastOp nad zastupnikom DS_i; inače, novi MTC samo prelazi na sledeću iteraciju petlje.

3.2.5 Testiranje

U ovom odeljku je ukratko objašnjen postupak koji je korišćen za testiranje DPSTM v3, bez ulaženja u detalje koji idu van okvira disertacije. Uopšteno govoreći, testiranje DPSTM v3 je izvedeno na tri nivoa testiranja, po principu od dole na gore, tj. od najnižeg ka najvišem nivou apstrakcije, u slojevitoj arhitekturi DPSTM v3.

Na prvom (najnižem) nivou testiranja su korišćeni testovi pojedinačnih modula i objekata (klasa), kraće nazivani *jedinični testovi* (eng. unit tests), koji su nasleđeni i prošireni od prethodne DPSTM v2. Ovi jedinični testovi su u suštini jednostavne teoreme sa jednom ili više tautologija, a testiranje zasnovano na jediničnim testovima je oblik automatskog dokazivanja teorema. Tautologije unutar jediničnih testova su izrazi koji proističu iz polazne specifikacije, odnosno funkcionalnih zahteva postavljenih pred projektovani sistem. Izvršilac (eng. test executor) jediničnih testova automatski izvršava svaki jedinični test u sledeća tri koraka: (1) poziva proceduru za pravljenje i postavljanje test objekata, (2) poziva sam jedinični test, u kom su sadržani iskazi akcija nad test objektima i tvrdnje (eng. assert) da su tautologije važeće, i (3) poziva proceduru za uništavanje test objekata. Na kraju se formira izveštaj o testiranju.

Na ovaj način su testirani svi moduli i objekti u DPSTM v3. Tipičan primer jediničnog testa za DPSTM API se sastoji od sledećih koraka: (1) pozovi API funkciju addVars radi dodavanja nove t-promenljive x, (2) tvrdi da je povratna vrednost addVars jednaka „yes“, (3) pozovi API funkciju putVars radi postavljanja x na vrednost 10, (4) tvrdi da je povratna vrednost putVars jednaka „yes“, itd.

Na drugom nivou testiranja DPSTM v3 je korićeno poluautomatsko testiranje po principu bele kutije (eng. white-box testing) u kom je objekat testiranja potpuno poznat i koji služi za proveru internog ponašanja objekta testiranja. U ovoj formi testiranja je dozvoljeno dodavanje namenskog programskog koda za potrebe testiranja, kao što je na primer programski kod koji simulira nastanak otkaza u sistemu. Takav kod se obično aktivira ručnom akcijom osobe koja izvodi testiranje.

Glavni cilj na drugom nivou testiranja DPSTM v3 je bila verifikacija glavnih doprinosa ove doktorske disertacije (distribuirana MS-FSM, procedura za prevazilaženje otkaza, prilagođeni replikacioni protokol, i procedura za oporavak servisa nakon otkaza), koja je urađena kroz poluautomatsko testiranje celog sistema po principu bele kutije, sa ciljem da se pokriju sve putanje kroz funkciju doOp (vidi

odeljak 3.2.2). U tom cilju su izvedena četiri testa koji pokrivaju četiri putanje kroz funkciju doOp. Pošto je DPSTM v3 uspešno prošla ove testove, može se zaključiti da su osnovni doprinosi ove doktorske disertacije na ovaj način verifikovani.

Na trećem nivou testiranja je izvedeno testiranje celog sistema (tj. sistemsko testiranje) radi njegove validacije i eksperimentalne evaluacije u laboratorijskim uslovima, što je detaljno objašnjeno u narednom odeljku (Odeljak 3.3).

3.3 Eksperimentalna evaluacija DPSTM v3

Ovaj odeljak pretstavlja eksperimentalnu evaluaciju DPSTM v3, zasnovanu na merenju propusnosti sistema, koja je ovde definisana kao broj transakcija u jedninici vremena, za šest različitih radnih opterećenja. Ova radna opterećenja su različite mešavine dve vrste primitivnih transakcija, nazvane *transakcije čitanja* i *transakcije upisa*. Kao što njihova imena sugerišu, prve izvode operaciju čitanja (koristeći getVars), dok druge izvode operaciju upisa (koristeći putVars). Eksperimentalna evaluacija je urađena u obliku testa sa poređenjem performanse dve tipične mrežne konfiguracije, nazvane potpuno distribuirana mrežna konfiguracija (u kojoj se svaka komponenta sistema izvršava na zasebnoj mašini) i mrežna konfiguracija sa kolokacijom (u kojoj su aplikacioni procesi i njihovi lokalni serveri podataka kolocirani u istim mašinama).

Dva glavna cilja eksperimentalne evaluacije su da se demonstrira: (1) da se propusnost sistema povećava super-linearno (tj. linearno ili bolje od toga) kako se udeo operacija čitanja povećava od 0% do 100% (i udeo operacija upisa simetrično smanjuje od 100% do 0%), i (2) da je propusnost sistema za mrežnu konfiguraciju sa kolokacijom mnogo bolja od propusnosti sistema za potpuno distribuiranu mrežnu konfiguraciju. Rezultati eksperimentalne evaluacije u Odeljku 3.3.4 (rezultati i diskusija) pokazuju da su oba cilja uspešno ostvarena.

3.3.1 Test aplikacija

Test aplikacija je sama po sebi tipična DPSTM aplikacija, koja se sastoji od vodećeg procesa i grupe od pet procesa radnika. Vodeći proces izvodi inicijalizaciju sistema, orkestraciju sistema, i izračunavanje i raportiranje konačnih rezultata testa (propusnost sistema i njenu standardnu devijaciju). S druge strane, svaki proces radnik

izvodi svoj udeo zadatog radnog opterećenja, računa svoju individualnu propusnost, i vraća svoj rezultat vodećem procesu.

Vodeći proces se sinhronizuje i komunicira sa procesima radnicima preko DPSTM. Vodeći proces i procesi radnici se sinhronizuju korišćenjem mehanizma asimetrične barijere zasnovanom na DPSTM, koji je realizovan pomoću dve funkcije, pod nazivom masterBarrier i workerBarrier. Kao što njihova imena sugerišu, prvu funkciju koristi vodeći proces, dok drugu koriste procesi radnici. Pored sinhronizacije, DPSTM se koristi od strane procesa radnika za isporuku svojih rezulata, upisom rezultata u odgovarajuće t-promenljive, i od strane vodećeg procesa za prikupljanje rezulata procesa radnika, čitanjem rezultata iz odgovarajućih t-promenljivih.

Prema globalno poznatom rasporedu, test aplikacija sprovodi šest faza koje odgovaraju različitim radnim opterećenjima, tj. mešavinama transakcija čitanja i upisa. Ova radna opterećenja su sledeća: (1) 0% čitanja plus 100% upisa, (2) 20% čitanja plus 80% upisa, (3) 40% čitanja plus 60% upisa, (4) 60% čitanja plus 40% upisa, (5) 80% čitanja plus 20% upisa, i (6) 100% čitanja plus 0% upisa.

U svakoj fazi, proces radnik radi bilo kao proces *čitalac* koji izvodi transakciju čitanja ili kao proces *pisac* koji izvodi transakciju upisa. Tako je alternativni način da se specificiraju gore izlistana radna opterećenja, da se specificira broj procesa čitalaca i broj procesa pisaca, za svaku fazu. Na primer, radno opterećenje br. 3, sa 40% čitanja i 60% upisa, odgovara fazi sa dva procesa čitaoca i tri procesa pisaca.

Dva najvažnija parametra koje koristi test aplikacija su NTXNS i NMESR. NTXNS je broj transakcija čitanja ili upisa koje izvodi pojedinačni proces radnik u svakoj fazi (u ovom radu, ovaj parametar je bio postavljen na 10000). NMESR je broj merenja vremena izvršenja uzetih u svakoj fazi (tj. za odgovarajuće radno opterećenje), koja se dalje koriste za izračunavanje prosečne propusnosti za svaku fazu (u ovom radu, ovaj parametar je bio postavljen na 3).

Izvršenje svake faze je organizovano na sledeći način. Vodeći proces poziva funkciju masterBarrier dva puta za redom – prvi put da bi sačekao sve procese radnike i da bi im zatim signalizirao početak faze, i drugi put da bi sačekao da procesi radnici završe fazu. Po povratku iz drugog poziva masterBarrier, vodeći proces računa statističke podatke za tu fazu. S druge strane, svaki proces radnik prvo poziva funkciju workerBarrier da bi sačekao početni signal od vodećeg procesa, zatim izvodi svoj udeo

radnog opterećenja u tekućoj fazi (tj. izvodi NMESR paketa od po NTXNS transakcija), i poziva funkciju workerBarrier da bi signalizirao da je završio svoj zadatak. Na kraju šeste faze, vodeći proces skladišti i raportira konačne rezultate.

3.3.2 Test mrežne konfiguracije

Kao što je već spomenuto, eksperimentalna evaluacija je urađena na dve test mrežne konfiguracije – potpuno distribuiranoj mrežnoj konfiguraciji (PDMK) i mrežnoj konfiguraciji sa kolokacijom (MKSK). Obe test mrežne konfiguracije sadrže: dva DS (DS_1 i DS_2), dva TC (TC_1 i TC_2), jedan vodeći proces test aplikacije (M), i pet procesa radnika test aplikacije (W_1 do W_5) – sve zajedno deset sistemskih komponenti. U cilju balanisranja opterećenja, u obe test mrežne konfiguracije, M, W_2, i W_4 korsite DS_1 kao svoj lokalni DS, dok W_1, W_3, i W_5 koriste DS_2 kao svoj lokalni DS.

Glavna razlika između ove dve test mrežne konfiguracije je u načinu kako je tih deset sistemskih komponenti raspoređeno na zasebne mašine u ciljnoj fizičkoj mreži. U PDMK, svaka komponenta sistema je raspoređena na svoju sopstvenu mašinu (tj. deset komponenti je raspoređeno na deset zasebnih mašina), dok su u MKSK, procesi test aplikacije kolocirani sa svojim lokalnim serverima podataka a TC serveri su raspoređeni na svojim sopstvenim mašinama. Dakle, u MKSK se sve zajedno koriste četiri mašine – dve mašine za dva TC, jedna mašina za DS_1, M, W_2, i W_4, i jedna mašina za DS_2, W_1, W_3, i W_5.

3.3.3 Eksperimentalna postavka

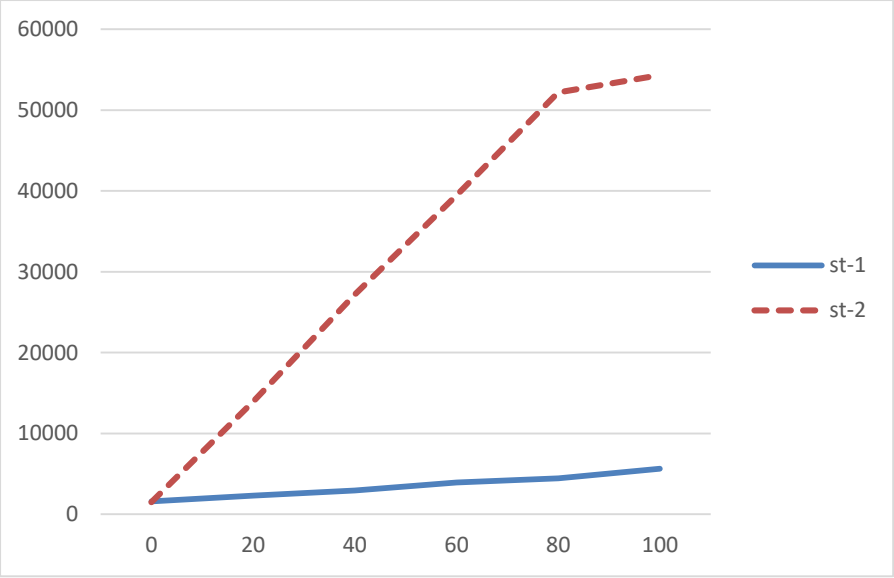
Karakteristike eksperimentalne postavke su sledeće. LAN karakteristike: utičnice računara u laboratoriji su povezane na jedan komutator Cisco Catalyst WS-C2960-48TT-L. Ovaj komutator obezbeđuje 10/100Mbps korisničke pristupne priključke i vezu na gore brzine 1 Gbps (ova veza nije korišćena u test aplikaciji). Mrežni kablovi su UTP kategorije CAT5e. Karakteristike računara: CPU - Intel Corei7-7700 CPU na 3.60GHz (4 jezgra x64), MB - Asus Prime B250M-C, RAM - 1 modul x 16 GB DDR4 na 2133 MHz, HDD - 2TB (Toshiba), Graphics - Intel(R) HD Graphics 630. Verzije softvera: OS - Windows 10 Professional (x64). Verzija Pajtona 3.7.3 (Datum izdanja: 25 mart 2019.) [MSC v.1916 64 bita (AMD64)] za win32.

3.3.4 Rezultati i diskusija

Eksperimentalni reulutati su dati u Tabeli 3 i ilustrovani na Slici 6. Vrste u Tabeli 3 odgovaraju različitim radnim opterećenjima, dok kolone u Tabeli 3 odgovaraju: udelu operacija čitanja (r), propusnosti sistema za PDMK (st-1), standardnoj devijaciji za st-1 (stdev-1), propusnosti sistema za MKSK (st-2), i standardnoj devijaciji za st-2 (stdev-2).

Tabela 3 – Eksperimentalni rezultati

r [%]	st-1 [txns/s]	stdev-1 [%]	st-2 [txns/s]	stdev-2 [%]
0	1579	0.23	1498	0.59
20	2320	0.75	13937	1.47
40	2949	0.28	27187	1.03
60	3920	3.90	39428	1.28
80	4432	1.90	52195	0.23
100	5636	1.45	54329	2.49



Slika 6 – Ilustracija eksperimentalnih rezultata iz Tabele 3

Apcisa na Slici 6 prikazuje vrednosti za udeo operacija čitanja (r), dok ordinata na Slici 6 prikazuje vrednosti za propusnost sistema u transakcijama po sekundi (txns/s). Puna kriva i isprekidana kriva redom prikazuju vrednosti za st-1 i st-2.

Vrednosti za st-1 u Tabeli 3 (puna kriva na Slici 6) ukazuju da se st-1 super-linearno povećava sa povećanjem r, od 1579 txns/s naviše do 5636 txns/s (pri čemu su vrednosti za stddev-1 sasvim prihvatljive). U Slučaju st-1, glavno ograničenje sistema je kapacitet komunikacione opreme.

Slično, vrednosti st-2 u Tabeli 3 (isprekidana kriva na Slici 6) ukazuju da se st-2 linearno povećava sa povećanjem r, naviše do vrednosti r = 80%, dok je rast st-2 u intervalu od r = 80% do r = 100% sporiji, zato što se za vrednost r = 80% dostiže pun kapacitet mašina, koje su domaćini za aplikacione procese i njihove lokalne servere podataka. Naime, ove mašine imaju samo četiri jezgra a treba da budu domaćini za četiri sistemske komponente plus za lokalni operativni sistem (i njegove procese). Iz tog razloga, nema dovoljno raspoloživih procesorskih jezgara da bi se sve sistemske komponente izvršavale paralelno, i stoga te mašine ne mogu da obezbede zahtevani servis po zahtevanom tempu. Jedna od stavki u planu za budući rad je ekpserimentalna evaluacija na mašinama sa dovoljno procesorskih jezgara, kako bi se definitivno potvrdila ova pretpostavka.

Konačno, poređenjem st-2 sa st-1, vidi se da su vrednosti za st-2 za red veličine veće, i time se potvrđuje prednost kolociranja aplikacionih procesa sa njihovim lokalnim serverima podataka.

3.3.5 Prednosti i ograničenja

U odeljku 2.4 je objašnjen pristup eksperimentalnoj evaluaciji korišćen u ovom istraživanju, koji je zasnovan na sintetičkim radnim opterećenjima, koja se prave kao mešavine read i write transakcija. U ovom odeljku su izložene prednosti i ograničenja tog pristupa iz perspektive konkretne eksperimentalne evaluacije urađene za DPSTM v3 u ovom poglavlju, polazeći najpre od prednosti.

Prva prednost je da ovaj pristup obezbeđuje dobro pokrivanje celog opsega mogućih aplikacija. U ovoj konkretnoj evaluaciji, korišćeno je šest različiti radnih opterećenja, što deluje da je ovde dovoljno zato što su krive na Slici 6 prilično glatke. U principu, moguće je uvesti još više radnih opterećenja ukoliko je potrebo dobiti još bolje pokrivanje celog opsega aplikacija.

Druga prednost je da se pomoću ovog pristupa mogu dobiti opšti trendovi, kao i donje i gornje granice performanse. Na primer, posmatranjem krivih na Slici 6, i podataka u Tabeli 3, može se videti da se propusnost sistema povećava sa povećanjem udela read operacija u radnom opterećenju. Ovo znači da se DPSTM v3 dobro ponaša, i u skladu sa očekivanjima.

Treća prednost je da se na osnovu rezultata ovog pristupa (dobijenih podataka) može proceniti performansa proizvoljnog radnog opterećenja korišćenjem interpolacije. Na primer, na osnovu vrednosti od st-2 za r = 40% i r = 60%, može se proceniti vrednost od st-2 za r = 50%, i napraviti predviđanje da bi očekivana vrednost od st-2 za r = 50% trebala da bude oko 33307 (read i write transakcija u sekundi). Moguće je praviti takve procene čak i za dinamička radna opterećenja koja menjaju svoje ponašanje tokom vremena, ako su raspoloživi njihovi profili, tj. ako je poznato kako se u tim radnim opterećenjima menja mešavina read i write operacija tokom vremena. Ova mogućnost je prilično važna za inženjering sistema zasnovanih na DPSTM v3, naročito kad se računa kapacitet sistema.

Sad prelazimo na ograničenja ove konkretne eksperimentalne evaluacije. Prvo ograničenje je da ona pokriva samo dve mrežne konfiguracije. Međutim, za te dve mrežne konfiguracije se pretpostavlja da one predstavljaju dva suprotna ekstrema u celom opsegu (tj. dimenziji) mogućih mrežnih konfiguracija, i da vrednosti propusnosti sistema za te dve konfiguracije pretstavljaju redom donju i gornju granicu za propusnost sistema. Ako je ova pretpostavka tačna, onda se može očekivati da će kriva koja predstavlja propusnost sistema za proizvoljnu mrežnu konfiguraciju biti negde između krivih st-1 i st-2 na Slici 6. Planirano je da se u budućem radu ova pretpostavka proveri putem evaluacije DPSTM v3 na dodatnim mrežnim konfiguracijama.

Drugo ograničenje ove konkretne mrežne evaluacije je da je ona urađena za jednu konkretnu eksperimentalnu postavku, korišćenjem jedne vrste hardvera i jedne mrežne tehnologije, konkretno u pitanju su PC računari sa procesorima sa četiri jezgara i komutirani žičani LAN sa pristupnim priključcima brzine 100Mbps. S druge strane, IoT sistemi obično koriste bežične mrežne tehnologije, kao što su WiFi, Zigbee, itd. Zbog toga je planirano da se u budućem radu DPSTM v3 evaluira na tim bežičnim mrežnim tehnologijama.

4. FORMALNA VERIFIKACIJA DPSTM V3

U ovom poglavlju je pretstavljena formalna verifikacija DPSTM (v3 se u ovom poglavlju podrazumeva). Skraćenica (D)PSTM označava „PSTM i DSPTM“. U narednih pet odeljaka su redom izloženi: opšti push/pull semantički model, interni algoritmi PSTM, interni algoritmi DPSTM, push/pull semantički model (D)PSTM i dokazi kriterijuma koretnosti (D)PSTM.

4.1 Push/Pull semantički model

U push/pull semantičkom modelu [37], stanje sistema je apstrahovano sa sledeća dva tipa loga: (1) jedan globalni (ili deljeni) log operacija koje su gurnule (PUSH) sve niti, i (2) po jedan lokalni log operacija za svaku nit. U lokalnom logu postoje dve vrste operacija: (1) operacije koje je nit povukla (PULL) iz globalnog loga, i (2) operacije koje je nit primenila (APPLY) lokalno. Ovaj model uključuje sledeća pravila:

- APPLY(*op*): Primeni operaciju *op* na lokalni log.
- UNAPPLY(*op*): Ukloni operaciju *op* iz lokalnog loga.
- PUSH(*op*): Gurni operaciju *op* iz lokalnog loga u globalni (*op* ostaje i u lokalnom logu).
- UNPUSH(*op*): Ukloni operaciju *op* iz globalnog loga.
- PULL(*op*): Povuci operaciju *op* iz globalnog loga u lokalni (*op* ostaje i u globalnom logu).

- UNPULL(*op*): Ukloni *op* iz lokalnog loga.
- CMT(*txn*): Završi transakciju *txn* (završetak može biti uspešan ili neuspešan).

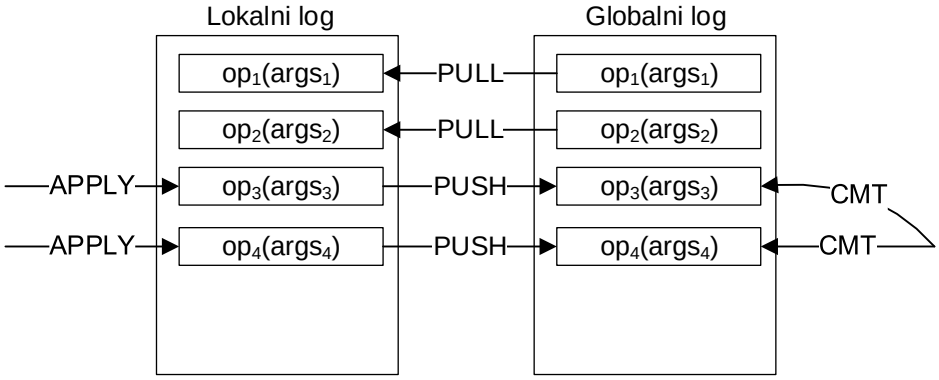
Generalno, nakon što transakcija (tj. nit koja je izvršava) primeni neku operaciju *op* na svoj lokalni log pomoću pravila APPLY, ona može gurnuti tu operaciju u globalni log pomoću pravila PUSH. Nakon što je neka operacija bila gurnuta u globalni log, druge transakcije mogu da povuku tu operaciju pomoću pravila PULL. Povlačenje neke operacije *op* iz globalnog loga znači primenu te operacije na lokalni log. Transakcije mogu povlačiti iz globalnog loga i uspešno završene operacije (eng. committed) i operacije koje su u toku, tj. koje još nisu završene (eng. uncommitted). Napomena: (D)PSTM u svom globalnom logu ima samo uspešno završene operacije.

U push/pull semantičkom modelu postoje i inverzna pravila. Pravilo UNPULL odbacuje operaciju iz lokalnog loga koja je predhodno bila povučena iz globalnog loga. Pravilo UNPUSH uklanja operaciju iz globalnog loga koja je u njega predhodno bila gurnuta. Pravilo UNAPPLY vraća lokalno stanje transakcije unazad (eng. rewind) uklanjanjem predhodno primenjene operacije iz lokalnog loga. Svaka transakcija se završava pomoću pravila CMT, koje zahteva da: (1) predhodno budu gurnute sve operacije čiji efekti treba da budu globalni, i (2) da su sve povučene operacije u međuvremenu uspešno završene (eng. committed).

Različiti transakcioni algoritmi koriste različite kombinacije push/pull pravila. Pesimistički algoritmi guraju (PUSH) operacije odmah nakon njihove lokalne primene (APPLY), optimistički algoritmi kao što je (D)PSTM guraju (PUSH) sve svoje operacije na samom kraju (CMT), dok hibridni algoritmi koriste neku mešavinu predhodna dva pristupa. Netransparentni (eng. opaque) algoritmi, koji zadovoljavaju svojstvo netransparentnosti (eng. opacity), ne smeju da povlače operacije koje još nisu završene (eng. uncommitted), dok je transparentnim algoritmima (eng. non-opaque) dozvoljeno da povlače i operacije koje još nisu završene.

Slika 7 ilustruje jednostavan primer push/pull transakcije za prenos novca, koja se koristi kao dežurni primer u ovom poglavlju. Zadatak ove transakcije je da realizuje plaćanje od *p* eura, sa računa *x* na račun *y*, oduzimanjem *p* eura sa računa *x* i dodavanjem *p* eura na račun *y*. Da bi realizovala ovaj zadatak, ova transakcija izvodi sledeće korake: (1) povuci operacije op_1 i op_2 koje su bile korišćene za postavljanje tekućih vrednosti transakcionih promenljivih (koje se skraćeno zovu: t-promenljive) *x* i

y, respektivno, (2) primeni operaciju op_3 koja postavlja novu vrednost od *x* na (*x* – *p*), (3) primeni operaciju op_4 koja postavlja novu vrednost promenljive *y* na (*y* + *p*), (4) gurni operacije op_3 i op_4 iz svog lokalnog loga u globalni log, i (5) završi transakciju, tj. označi gurnute operacije op_3 i op_4 u globalnom logu kao uspešno završene (eng. committed).



Slika 7 – Push/Pull transakcija za prenos novca

Dalje sledi definicija četiri (od sedam) pravila koja su korišćena za modeliranje (D)PSTM semantike, a to su pravila: APPLY, PUSH, PULL i CMT [37].

Definicija 1 (Relevantna push/pull pravila): Push/Pull semantička pravila relevantna za (D)PSTM su definisana na sledeći način:

$$\frac{\begin{array}{cc}(i) & c_1 \downarrow_{\text{tx}} (m, c_2) \\ (ii) & L_1 \text{ allows } \langle m, \sigma_1, \sigma_2 \rangle \\ (iii) & \text{fresh}(id)\end{array}}{\begin{array}{c}\{\text{tx } c_1, \sigma_1, L_1\}, G_1 \rightarrow^{\text{fwd}} \\ \{\text{tx } c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id \rangle, \text{unpushed } c_1]\}, G_1\end{array}} \text{APPLY}$$

$$\frac{\begin{array}{cc}(i) & op \lhd \lfloor L_1 \rfloor_{\text{unpushed}} \\ (ii) & \lfloor G_1 \rfloor_{\text{gUCmt}} \backslash \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \lhd op \\ (iii) & G_1 \text{ allows } op\end{array}}{\begin{array}{c}\{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{unpushed } c_2] \cdot L_2\}, G_1 \rightarrow^{\text{fwd}} \\ \{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{pushed } c_2] \cdot L_2\}, G_1 \cdot [op, \text{gUCmt}]\end{array}} \text{PUSH}$$

$$\frac{\begin{array}{cc}(i) & op \notin L \\ (ii) & L \text{ allows } op \\ (iii) & op \lhd \lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}\end{array}}{\begin{array}{c}\{\text{tx } c_1, \sigma_1, L\}, G_1 \cdot [op, g] \cdot G_2 \rightarrow^{\text{fwd}} \\ \{\text{tx } c_1, \sigma_1, L \cdot [op, \text{pulled}]\}, G_1 \cdot [op, g] \cdot G_2\end{array}} \text{PULL}$$

$$\frac{\begin{array}{cc}(i) & \mathrm{fin}(c)\\ (ii) & L_1 \subseteq G_1\\ (iii) & \forall op \in \lfloor L_1 \rfloor_{\mathrm{pulled}}, (op, g) \in G_1 . g = gCmt\\ (iv) & \mathrm{cmt}(G_1, L_1, G_2)\end{array}}{\begin{array}{c}T_1 \cdot \{(\mathrm{tx}\, c, c_1), \sigma, L_1\} \cdot T_2, G_1 \rightsquigarrow\\ T_1 \cdot \{c_1, \sigma, L_1\} \cdot T_2, G_2\end{array}}\ \mathrm{CMT}$$

$$\mathrm{cmt}(G_1, L_1, G_2) \equiv$$
$$G_2 = \mathrm{map}\left(\lambda(op, g) \cdot \begin{cases}(op, gCmt) & \text{if } op \in \lfloor L_1 \rfloor_{\mathrm{pushed}}\\ (op, g) & \text{otherwise}\end{cases}\right) G_1$$

U Definiciji 1 je za operator prebacivanja-ulevo (eng. left-mover operator) korišćen simbol '◁', zbog toga što simbol '◀' koji je korišćen u [37] ne postoji u MS Word editoru jednačina. Takođe, autori [37] su napravili grešku u CMT u [37], koja je ovde ispravljena na osnovu njihovog detaljnog tehničkog izveštaja, vidi [47].

Prema autorima [37], kriterijumi korektnosti za dato push/pull pravilo definišu uslove pod kojima se to pravilo može primeniti. U suštini, kriterijumi korektnosti za dato pravilo su premise, ili preduslovi, za to pravilo. Za svako pravilo u Definiciji 1, njegovi preduslovi, ili kriterijumi korektnosti (ovo su sinonimi), su izlistani iznad horizontalne linije unutar pravila. Tako na primer, prvi preduslov, označen sa (*i*), za pravilo APPLY je: $c_1 \downarrow_{\mathrm{tx}} (m, c_2)$, i tako redom.

U nastavku će ukratko biti objašnjeni pojmovi koji se koriste unutar push/pull pravila u Definiciji 1.

Programske niti izvršavaju programski kod *c*, koji uključuje pokretanje novih niti (fork), ažuriranje lokalnog steka (local_R), transakcije (tx *c*), metode *m* iz skupa metoda *M*, i iskaz za završetak niti (skip). Lokalni stek preko prostora Σ modelira argumente i povratne vrednosti metoda. Ažuriranja lokalnog steka uključuju relaciju $R \subseteq \Sigma \times \Sigma$. Redukcija unutar transakcije $c{\downarrow_{\mathbf{tx}}}(m, c')$ znači da se programski kod *c* može redukovati na (*m, c'*), gde je *m* sledeća metoda do koje se može doći a *c'* je ostatak koda u nastavku (posle *m*). Predikat fin(*c*) je istinit ako postoji redukcija od *c* na skip (tj. sledeći iskaz za izvršenje je skip).

Stanje je opisano logovima operacija. Operacija *op* je tupl $\langle m, \sigma_1, \sigma_2, id \rangle$, koji sadrži ime metode *m*, argumente metode σ_1, povratne vrednosti metode σ_2, i jedinstven

identifikator *id*. Predikat fresh(*id*) je istinit ako je *id* globalno jedinstven. Notacije $l_1 \cdot l_2$ i $l \cdot op$, gde su l_1, l_2, i l uređene liste operacija (logovi), znače: dodaj l_2 na l_1 i dodaj *op* na *l*, respektivno.

Predikat zatvoren na prefiksima (eng. prefix closed) "allowed *l*" ukazuje na to da li log operacija l odgovara nekom stanju. Alternativno, može se pisati "l allows $\langle m, \sigma_1, \sigma_2, id\rangle$", što znači "allowed $l \cdot \langle m, \sigma_1, \sigma_2, id\rangle$". Push/Pull pravila koraka (eng. step rules) se odnose na nit koja izvršava transakciju tx *c*, i ta pravila rukuju lokalnim stekom, lokalnim logom, i globalnim logom. Lokalni log je lista operacija zajedno sa njihovim status flegom *f*, gde *f* može biti: unpushed *c*, pushed *c*, i pulled (flegovi unpushed i pushed skladište programski kod *c* koji je bio aktivan u trenutku kada je odgovarajući slog loga bio napravljen). Globalni log je lista operacija zajedno sa statusnim flegom *g*, koji ukazuje na to da li je operacija završena (eng. committed) ili ne: *g* :: = gUCmt | gCmt.

Unutar push/pull pravila se koriste sledeća podizanja operacija nad skupovima na liste:

$$\langle m_1, \sigma_1, \sigma_1', id_1\rangle \in L \equiv \exists i.\, L[i].\mathrm{id} = id_1$$
$$G \backslash L \equiv \mathrm{filter}(\lambda(op, g).\, op \notin L) G$$
$$L \subseteq G \equiv \forall i.\, L[i].op \in G$$

U definicijama iznad, $L[i]$ se odnosi na *i*-ti element od *L*, $L[i].op$ se odnosi na *i*-ti operacioni tupl, a $L[i].id$ se odnosi na identifikator *i*-te operacije. Redosled u $G \setminus L$ se određuje na osnovu redosleda u *G*.

Operator prebacivanja ulevo $\lhd$ (eng. left-mover) nad logovima se definiše na sledeći način:

$$op_2 \lhd op_1 \equiv \forall l.\, l \cdot \{op_1, op_2\} \preccurlyeq l \cdot \{op_2, op_1\}$$

Neformalno, operacija op_2 se može prebaciti sa leve strane operacije op_1 ako kad god je $op_1 \cdot op_2$ dozvoljeno (allowed), i $op_2 \cdot op_1$ je takođe dozvoljeno i rezultujući log je isti (za definiciju operatora $\preccurlyeq$ vidi [17]). Podizanje operatora $\lhd$ na liste je definisano na sledeći način:

$$L_1 \lhd L_2 \equiv \forall op_1 \in L_1, op_2 \in L_2.\, op_1 \lhd op_2$$

Projekcija loga *L* na status fleg *f* je definisana na sledeći način:

$$\lfloor L \rfloor_f \equiv \mathrm{map\ fst}\ (\mathrm{filter}(\lambda(op, f').\, f = f') L)$$

Da bi se izbeglo ponavljanje, preduslovi pravila iz Definicije 1 su objašnjeni u okviru dokaza da su oni zadovoljeni (vidi odeljak 4.5). Ovde na kraju ovog odeljka su ukratko objašnjeni zaključci za pravila iz Definicije 1 (zaključak pravila se nalazi ispod horizontalne linije unutar pravila).

Generalno, zaključak kod svih pravila iz Definicije 1 je neka relacija, koja specificira prelaz iz tekućeg u naredno stanje. U slučaju prva tri pravila (APPLY, PUSH i PULL) koristi se relacija $\rightarrow^{\text{fwd}}$, kod koje je stanje (i tekuće, s leve strane, i naredno, s desne strane) zadato u obliku {tx *c*, *σ*, *L*}, *G*, gde su: tx *c* aktivan programski kod, *σ* je lokalni stek, *L* je lokalni log, i *G* je globalni log. Na primer, tekuće stanje u zaključku pravila APPLY je $\{\text{tx } c_1, \sigma_1, L_1\}, G_1$, što znači da je aktivan kod tx c_1, lokalni stek je σ_1, lokalni log je L_1, i globalni log je G_1.

Zaključak pravila APPLY glasi:

$$\{\text{tx } c_1, \sigma_1, L_1\}, G_1 \rightarrow^{\text{fwd}} \{\text{tx } c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \text{unpushed } c_1]\}, G_1$$

Značenje gornjeg zaključka je da u narednom stanju (s desne strane relacije, koja je prikazana u drugom redu): aktivan programski kod postaje tx c_2, lokalni stek postaje σ_2, i na kraj lokalnog loga se dodaje operacija $\langle m, \sigma_1, \sigma_2, id\rangle$ sa status flegom $\text{unpushed } c_1$.

Zaključak pravila PUSH glasi:

$$\{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{unpushed } c_2] \cdot L_2\}, G_1 \rightarrow^{\text{fwd}} \{\text{tx } c_1, \sigma_1, L_1 \cdot [op, \text{pushed } c_2] \cdot L_2\}, G_1 \cdot [op, \text{gUCmt}]$$

U tekućem stanju gornjeg zaključka, u lokalnom logu je operacija *op*, sa status flegom unpushed c_2, koja treba da se gurne u globalni log. U narednom stanju, istovremeno se menja status fleg operacije *op*, u lokalnom logu, na vrednost pushed c_2, i operacija *op* se dodaje na kraj globalnog loga sa status flegom gUCmt.

Zaključak pravila PULL glasi:

$$\{\text{tx } c_1, \sigma_1, L\}, G_1 \cdot [op, g] \cdot G_2 \rightarrow^{\text{fwd}} \{\text{tx } c_1, \sigma_1, L \cdot [op, \text{pulled}]\}, G_1 \cdot [op, g] \cdot G_2$$

U tekućem stanju gornjeg zaključka, u globalnom logu je operacija *op*, sa status flegom *g*, koja treba da bude povučena u lokalni log. U narednom stanju, operacija *op*, sa status flegom pulled, se dodaje na kraj lokalnog loga.

U slučaju pravila CMT koristi se relacija $\rightsquigarrow$, kod koje je stanje (i tekuće, s leve strane, i naredno, s desne strane) zadato u obliku *T*, *G*, gde je *T* niz programskih niti T_i

spojenih pomoću operatora · , a G je globalni log. Posmatrana nit je data u obliku $\{c, \sigma, L\}$, gde su: c aktivan programski kod, σ je lokalni stek, i L je lokalni log. Na primer, tekuće stanje u zaključku pravila CMT je $T_1 \cdot \{(\text{tx}\, c, c_1), \sigma, L_1\} \cdot T_2, G_1$, što znači da je za posmatranu nit aktivan kod $(\text{tx}\, c, c_1)$, lokalni stek je σ, lokalni log je L_1, i globalni log je G_1. Zapis $(\text{tx}\, c, c_1)$ označava da nit treba sekvecijalno da izvrši najpre transakciju tx c, a zatim programski kod c_1.

Zaključak pravila CMT glasi:

$$T_1 \cdot \{(\text{tx}\, c, c_1), \sigma, L_1\} \cdot T_2, G_1 \rightsquigarrow T_1 \cdot \{c_1, \sigma, L_1\} \cdot T_2, G_2$$

U narednom stanju gornjeg zaključka (koje je prikazano u drugom redu), posmatrana programska nit je prešla na izvršenje programskog koda c_1, a globalni log G_2 je dobijen pomoću predikata cmt(G_1, L_1, G_2), koji preslikava G_1 na G_2 tako što prepisuje operacije iz G_1 u G_2, jednu po jednu, pri čemu za sve operacije iz G_1, koje su gurnute iz lokalnog loga L_1, menja status flag na gCmt, tj. završava ih (eng. commited), a ostale operacije samo prepisuje, vidi Definiciju 1.

4.2 Algoritmi PSTM

Arhitektura sistema zasnovanog na PSTM je tipična klijent-server arhitektura realizovana u programskom jeziku Python [25]. Klijenti su aplikacione transakcije, koje zahtevaju usluge od PSTM (server) pozivajući PSTM API funkcije. PSTM poslužuje zahteve transakcija održavajući sistemski rečnik deljenih promenljivih D (koji odgovara globalnom logu u push/pull semantičkom modelu). Svi zahtevi transakcija se šalju ka PSTM preko FIFO reda, q, dok se odgovori od PSTM šalju nazad transakcijama preko njihovih cevi (postoji po jedna cev od PSTM do svake transakcije).

PSTM transakcija tipično pribavlja (eng. get) potrebne t-promenljive od PSTM i skladišti ih u svoje lokalne kopije t-promenljivih (koje odgovaraju lokalnom logu u push/pull semantičkom modelu), radi neku obradu podataka, uključujući ažuriranja lokalnih kopija t-promenljivih, i na kraju zahteva od PSTM završetak svih svojih operacija upisa nad deljenim t-promenljivama (eng. commit). PSTM poslužuje sve zahteve transakcija atomično (neprekidivo), jedan po jedan, čime obezbeđuje mogućnost serijalizacije uspešno završenih operacija ažuriranja.

Stavka sistemskog rečnika se definiše kao par (*ikey*, *ival*), gde su *ikey* i *ival* ključ i vrednost odgovarajuće t-promenljive, respektivno. Operaciju čitanja vrednosti stavke sa ključem *ikey* iz rečnika *D* zapisujemo kao *D*[*ikey*] ili read(*ikey*), a operaciju upisa nove vrednosti *ival* u stavku sa ključem *ikey* u rečniku *D* zapisujemo kao *D*[*ikey*] := *ival* ili write(*ikey*, *ival*).

T-promenljiva se definiše kao trojka (*key*, *ver*, *val*), gde su *key*, *ver*, i *val*, ključ, verzija i vrednost t-promenljive. T-promenljiva (*key*, *ver*, *val*) se smešta kao stavka (*key*, (*ver*, *val*)) u sistemski rečnik *D*, čiji *ikey* i *ival* su *ikey* = *key* i *ival* = (*ver*, *val*), respektivno.

Glavne funkcije koje obezbeđuje PSTM API su sledeće:

- addVars(*q*, *K*) / *v*
- putVars(*q*, *W*) / *v*
- getVars(*q*, *K*) / *V*
- commitVars(*q*, *C*) / *v*

Generalno, funkcije addVars i putVars se uglavnom koriste u glavom procesu (main) aplikacije, dok se funkcije getVars i commitVars uglavom koriste u transakcionim procesima unutar aplikacije. Tipično, glavni proces koristi niz poziva addVars i putVars da formira (create) i inicijalizuje deljene t-promenljive. S druge strane, transakcija tipično na svom početku koristi poziv getVars da pribavi svoje lokalne kopije potrebnih t-promenljivih, dalje radi neku obradu podataka, i na kraju koristi poziv commitVars da završi sva svoja ažuriranja (commit). Dalje sledi detaljniji opis ovih API funkcija.

Arqument *q* je FIFO red koji povezuje transakcione procese i PSTM serverski proces. Argument *K* je lista ključeva t-promenljivih. Argument *R* je lista t-promenljivih čije vrednosti su očitavane. Argument *W* je lista t-promenljivih u koje treba obaviti upise (tj. koje treba ažurirati). *R* i *W* liste se skraćeno zovu i read i write liste t-promenljivih. Argument C je lista ⟨ *R*, *W* ⟩, gde su *R* i *W* read i write liste t-promenljivih, respektivno. Povratna vrednost *v* je string koji ukazuje na to da li se funkcija uspešno završila ili ne: *v* ::= ‘yes’ | ‘no’. Povratna vrednost *V* je lista parova (*e*, *vv*), gde je *e* zastavica: e ::= true | false, koja ukazuje na to da li dati *key* iz *K* postoji u *D* ili ne, dok je *vv* par (*ver*, *val*) ako je *e* = true ili none ako je *e* = false.

Kao što je već rečeno, PSTM server poslužuje PSTM API pozive sekvencijalno pozivjući svoje interne funkcije sa istim imenom ali bez prvog argumenta *q*. U nastavku su data kratka objašnjenja i pseudokodovi ovih internih funkcija, vidi Algoritme 2 do 5, respektivno.

Funkcija addVars dodaje t-promenljive čiji ključevi su u *K* postavljajući odgovarajuće *D* stavke na podrazumevane vrednosti (0, none), vidi Algoritam 2.

Algoritam 2 – Ineterna funkcija PSTM servera addVars

```
1: addVars(K)
2:   for k in K
3:     D[k] := (0, none)
4:   return 'yes'
```

Funkcija putVars postavlja inicijalne vrednosti t-promenljivih kako je specificirano sa *W*. Preciznije, funkcija putVars(*W*) se redukuje na poziv commitVars(*q*, ⟨ ⟨⟩, *W* ⟩), tj. ona radi kao funkcija commitVars sa praznom read listom, vidi Algoritam 3.

Algoritam 3 – Interna funkcija PSTM servera putVars

```
1: putVars(W)
2:   return commitVars([ [], W ])
```

Funkcija getVars vraća listu *V* za zadatu listu *K*. Funkcija radi na sledeći način. Inicijalno, postavi *V* na praznu listu ⟨⟩. Zatim za svako *key* ∈ *K*, proveri da li je *key* ∈ *D*, i ako jeste, dodaj par (true, (*ver*, *val*)) na kraj *V*, inače dodaj (false, none) na kraj *V*.

Algoritam 4 – Interna funkcija PSTM servera getVars

```
1: getVars(K)
2:   V := []  // postavi V na praznu listu
3:   for k in K
4:     vv := D[k]
5:     if vv = none then v := (false, none)
6:     else v := (true, vv)
7:     V := V + v  // dodaj v u V
8:   return V
```

Funkcija commitVars obavlja sva ažuriranja kako je specificirano sa *C*, i vraća 'yes' ako je ovo bilo uspešno, inače vraća 'no' ukazujući da je završetak transakcije neuspešan (eng. aborted). Funkcija radi na sledeći način. Prvo, proveri da li su verzije svih t-promenljivih u *R* i *W* iste kao u *D*. Ako nisu, vrati 'no'. Ako jesu, ažuriraj sve t-promenljive u *W* povećanjem njihove verzije za 1 i postavljanjem njihove nove vrednosti kako je specificirano sa *W*, i na kraju vrati 'yes', vidi Algoritam 5.

Algoritam 5 – Interna funkcija PSTM servera commitVars

```
1: commitVars(q, C)
2:   R, W := C
3:   if for all t in (R union W). t.ver = D[t.key].ver then
4:     for w in W
5:       write( w.key, (D[w.key].ver + 1, w.val) )
6:     return 'yes'  // transakcija se uspešno završila
7:   else
8:     return 'no'  // transakcija se neuspešno završila
```

U Algoritmima 4 i 5, operacije čitanja su zapisane implicitno kao *D*[*ikey*], dok su operacije upisa zapisane eksplicitno kao write(*ikey, ival*), vidi liniju 5 u Algoritmu 4. Ovo je urađeno sa namerom da se naglase write operacije nad *D*, jer je dovoljno koristiti samo write operacije (bez read) u push/pull semantičkom modelu (D)PSTM, vidi odeljak 4.1 u [28].

4.3 Algoritmi DPSTM

Arhitektura sistema zasnovanog na DPSTM v3 je nastala kroz niz evolutivnih proširenja osnovne arhitekure zasnovane na PSTM. U prvom koraku je nastala DPSTM v1 uvođenjem zastupnika transakcije (tj. DPSTM klijenta) radi podrške distribuiranih aplikacija sa transakcijama koje se izvršavaju na različitim mašinama u mreži. U drugom koraku je nastala DPSTM v2 uvođenjem transakcionog koordinatora (TC) i *n* replika DPSTM v1 (nazvanih serverima podataka, DS) radi: (1) poboljšanja performanse aplikacija koje dominantno čitaju t-promenljive i rade u režimu konačne konzistentnosti, kao i (2) otpornosti na otkaze servera podataka. U trećem koraku je nastala DPSTM v3 uvođenjem para transakcionih koordonatora koji rade u režimu

vodeći-prateći radi otpornosti na otkaze pojedinačnih transakcionih koordinatora, vidi Slike 1 i 2 u prethodnom poglavlju ove disertacije.

Radi jednostavnosti izlaganja, a bez gubitka opštosti, posmatra se sistem u kom nema otkaza servera, što znači da sve transakcije vodi vodeći transakcioni koordinator i da su sve replike idealno poravnate pa je dovoljno posmatrati baznu repliku. Drugo, s obzirom da je cilj dokazati da DPSTM v3 ima svojstvo mogućnost serijalizacije (tj. sekvencijalne konzistentnosti) posmatra se sistem u kom sve transakcije rade u režimu sekvencijalne konzistentnosti. Treće, pošto je DPSTM v3 najopštija od svih postojećih DPSTM (tj. od v1, v2, i v3), u ovom poglavlju skraćenica DPSTM se odnosi na nju. Četvrto, skraćenica DPSTM API se odnosi na API između transakcije i njenog zastupnika (taj API je isti u svim verzijama DPSTM) .

Kao što je u prethodnom poglavlju prikazano, DPSTM održava *n* replika istog sistemskog rečnika deljenih t-promenljivih, *D*, kao PSTM. Svaka t-promenljiva (*key*, *ver*, *val*) se i ovde skladišti u *D* kao (*ikey*, *ival*), gde je *ikey* jednak *key*, a *ival* je jednak paru (*ver*, *val*). DPSTM podržava isti skup funkcija nad *D* (addVars, putVars, itd.), i ona obavlja te operacije atomično, čime se obezbeđuje mogućnost serijalizacije DPSTM. Dalje, DPSTM transakcije imaju isto ponašanje, tj. isti životni ciklus, kao PSTM transakcije – one na početku pribavljaju svoje kopije t-promenljivih, zatim rade svoju lokalnu obradu podataka, i na kraju ažuriraju modifikovane t-promenljive.

Potrebno arhitektonsko proširenje je napravljeno uvođenjem dva nivoa zastupnika. DPSTM klijenti transparentno povezuju transakcije sa pojedinačnim TC, a DS klijenti povezuju TC sa pojedinačnom replikom (DS serverom), tako da transakcije nisu svesne da se one i DPSTM izvršavaju na različitim mašinama – transakcije pozivaju operacije nad DPSTM klijentima, koji dalje delegiraju te operacije vodećem TC, a TC onda te operacije delegira replikama (kao što je već detaljno objašnjeno u prethodnom poglavlju).

DPSTM je projektovana da ima API sa istom semantikom kao PSTM API kako bi se omogućio jednostavan prenos (eng. porting) već razvijenih softverskih komponenti zasnovanih na PSTM, kao što su konkurentne strukture podataka, na DPSTM. Jedina razlika između PSTM API i DPSTM API je njihova različita sintaksa – prva je funkcionalna (ili proceduralna), dok je druga objektno-orijentisana. PSTM API je definisan kao skup funkcija modula stm.py, dok je DPSTM API definisan kao

skup funkcija objekta DPSTM klijenta. Sve funkcije u PSTM API imaju svoje ekvivalente u DPSTM API.

Glavna sintaksna razlika između PSTM i DPSTM API je u sintaksi poziva API funkcija. Svaka funkcija u PSTM API ima PSTM red čekanja kao svoj prvi argument i poziva se kao jednostavna funkcija (nekog Python modula), dok ekvivalentna funkcija u DPSTM API nema red čekanja kao argument i poziva se nad objektom zastupnika (koji skriva mrežnu komunikaciju i DPSTM red čekanja). Formalno, poziv PSTM API funkcije *f*(*q, args*) je ekvivalentan pozivu DPSTM API funkcije *p.f*(*args*), gde je *q* PSTM red čekanja, *args* su drugi argumenti PSTM API funkcije *f*, i *p* je objekat zastupnika DPSTM. U oba slučaja funkcija *f* vraća istu vrednost *r*.

Unutar DPSTM, DPSTM klijent i TC su povezani preko tzv. eksternog DPSTM API, a DS klijent i DS su povezani preko tzv. internog DPSTM API. Svi ovi API su uz neznatne razlike isti. Eksterni DPSTM API je dobijen proširivanjem DPSTM (na kraj liste argumenata je dodata ID transakcije), dok je interni DPSTM API dobijen proširivanjem eksternog (dodavanjem dve nove funkcije za obezbeđenje otpornosti na otkaze transakcionog koordinatora), kao što je već detaljno objašnjeno u prethodnom poglavlju.

Međutim, realizacije ovih API su različite. Funkcije objekta DPSTM klijenta jednostavno delegiraju svoj posao funkciji objekta DPSTM, koji ustvari predstavlja vodećeg TC (pozivajući istoimenu funkciju nad objektom DPSTM, prosleđujući joj argumente, i vraćajući njenu povratnu vrednost). U sledećem koraku u lancu poziva, istoimena funkcija DS klijenta delegira svoj posao funkciji DS objekta. Na kraju, funkcija objekta DS efektivno obavlja zahtevanu operaciju nad svojim lokalnim *D*.

Konačno, arhitektura sistema zasnovanog na DPSTM je formalizovana u obliku pseudokodova implementacija ovih API, u kojima su radi jednostavnosti izostavljeni detalji vezani za otkaze servera. Implementacija DPSTM API u objektu DPSTM klijent je specificirana pseudokodom u Algoritmu 6, gde je *dpstm* objekt DPSTM (tj. objekt vodećeg TC). Implementacija eksternog DPSTM API u objektu DS klijent je specificirana pseudokodom u Algoritmu 7, gde je *DS* lista od *n* objekata DS servera, DS[1] je bazna replika. Implementacija internog DPSTM API u objektu DS definisana sa pseudokodovima u Algoritmima 2 do 5 (tj. interne funkcije unutar PSTM i DS ekvivalente). To znači da se inicijalni pozivi DPSTM API funkcija nad objektom

DPSTM klijent na kraju preslikavaju na odgovarajuće funkcije u Algoritmima 2 do 5, koje su objašenje u prethodnom odeljku 4.3.

Algoritam 6 – Realizacija DPSTM API u objektu DPSTM klijent

```
dpstm  // dpstm je objekt DPSTM, tj. objekt vodećeg TC

1: addVars(K)
2:   return dpstm.addVars(K, txnid)

3: putVars(W)
4:   return dpstm.putVars(W, txnid)

5: getVars(K)
6:   return dpstm.getVars(K, txnid)

7: commitVars(C)
8:   return dpstm.commitVars(C, txnid)
```

Algoritam 7 – Realizacija eksternog DPSTM API v3 u objektu DS klijent

```
DS  // DS je lista od n objekata DS servera, DS[1] je bazna replika

01: addVars(K, txnid)
02:   for i = 1 to n
03:     ret = DS[i].addVars(K, txnid)
04:   return ret

05: putVars(W, txnid)
06:   for i = 1 to n
07:     ret = DS[i].putVars(W, txnid)
08:   return ret

09: getVars(K, txnid)
10:   return DS[1].getVars(K, txnid)
```

```
11: commitVars (C, txnid)
12:   for i = 1 to n
13:     ret = DS[i]. commitVars (C, txnid)
14:   return ret
```

4.4 Push/Pull semantički model (D)PSTM

U ovom odeljku je prikazan push/pull semantički model korišćenjem DPSTM API sintakse (ovaj model je prvi put objavljen u [32]). Generički transakcioni algoritam nad (D)PSTM, T, se definiše kao blok koda koji započinje pozivom API funkcije getVars (koja povlači svaku t-promenljivu tačno jedanput), obavlja neku lokalnu obradu uključujući ažuriranje kopija t-promenljivih, i završava se pozivom API funkcije commitVars, vidi Algoritam 8. Pošto je T pravolinijski algoritam (bez petlji) on se uvek završava.

Algoritam 8 – Generički transakcioni algoritam nad (D)PSTM, T

```
1: T(K)
2:   V := dpstm.getVars(K)
3:   Lokalna obrada sa ažuriranjima lokalnih kopija t-promenljivih.
4:   v := dpstm.commitVars(C)
```

Napomena uz Algoritam 8: Ovaj algoritam se generalizuje na proizvoljne transakcije sa proizvoljnim brojem poziva getVars i commitVars, u proizvoljnom redosledu, ali radi jednostavnosti uzeta je ova njegova najjednostavnija forma. Dalje sledi definicija samog modela.

Definicija 2 (Push/Pull semantički model (D)PSTM): Push/Pull semantički model (D)PSTM je preslikavanje operacija iz generičkog transakcionog algoritma T (uključujući operacije iz funkcija u celom stablu poziva funkcija) na odgovarajuća push-pull pravila putem funkcije sm, koja je definisana sa sledeće tri poddefinicije:

Definicija 2.1 (Preslikavanje linije 2 u Algoritmu 8): Poziv funkcije getVars(*K*) se preslikava na sledeći način. Za svako *k* u *K*, operacija *D*[*k*] (linija 4 u Algoritmu 4) se preslikava na pravilo PULL koje se odnosi na poslednju write operaciju iz

globalnog loga *G*, koja je korišćena da se postavi vrednost *D*[*k*] (linija 5 u Algoritmu 5). Dakle, poziv funkcije getVars(*K*) se preslikava na |*K*| pravila PULL, gde je |*K*| veličina liste *K* (tj. broj elemenata u toj listi).

Definicija 2.2 (Preslikavanje linije 3 u Algoritmu 8): Ažuriranja lokalnih kopija t-promenljivih (tj. write operacije nad njima) se preslikavaju na pravila APPLY, po jedno pravilo za svaku write operaciju.

Definicija 2.3 (Preslikavanje linije 4 u Algoritmu 8): Poziv funkcije commitVars(*C*) se preslikava na sledeći način. Za svaku t-promenljivu *w* u *W*, odgovarajuća write operacija (linija 5 u Algoritmu 5) se preslikava na pravilo PUSH. Kraj for petlje u liniji 4, u Algoritmu 5, se preslikava na pravilo CMT. Dakle, poziv funkcije commitVars(*C*) se preslikava na |*W*| pravila PUSH i jedno pravilo CMT sve zajedno.

Radi pojašnjenja Definicije 2, zapišimo ulazni algoritam T kao ulaznu sekvencu (tj. listu) funkcija $T_i = [f_1, f_2, f_3]$, gde je f_1 = getVars(*K*), f_2 = update(*K*), i f_3 = commitVars(*C*), pri čemu funkcija update ažurira lokalne kopije t-promenljivih sa ključevima u *K*. Radi jednostavnosti, pretpostavimo da *K* ima samo element *k*. Onda se funkcija sm preslikava (tj. mapira) preko T_i da bi proizvela izlaznu sekvencu push/pull pravila $T_o = sm(T_i) = [r_1, r_2, r_3]$, gde $r_1 = sm(f_1)$ = PULL, $r_2 = sm(f_2)$ = APPLY, $r_3 = sm(f_3)$ = PUSH, CMT.

Definicija 3 (Sekvencijalna specifikacija logova): Lokalni i globalni logovi su logovi hronološki uređenih write operacija, i njihova sekvencijalna specifikacija je definisana tako da poziv getVars([*k*]) povlači poslednju write operaciju u *D*[*k*] i da se svaka read operacija nad kopijom t-promenljive, unutar lokalne obrade u transakciji, izračunava (tj. evaluira) u vrednost upisanu poslednjom write operacijom nad tom kopijom t-promenljive.

Da bi ilustrovali izvršenje algoritma T (Algoritam 8) vraćamo se na dežurni primer transakcije za prenos novca, koji sad posmatramo kao jednu instancu algoritma T, označenu sa T_m, čiji pseudokod je dat u tabeli Algoritam 9.

Slika 8 ilustruje izvršenje algoritma T_m (Algoritam 9) unutar push/pull semantičkog modela. Leva strana slike prikazuje lokalni log L_m od transakcije T_m (koja izvršava algoritam T_m), središnji deo slike prikazuje globalni log *G*, a desni deo slike prikazuje prelaze iz tekućeg stanja u naredno stanje. Redovi na Slici 8 prikazuju tekuće

stanje (lokalni i globalni log) i sledeći događaj koji izaziva prelaz. Evolucija sistema napreduje od gore na dole. Pravougaoni simboli (tj. kutijice) u logovima odgovaraju operacijama. Označene operacije u globalnom logu odgovaraju završenim operacijama (eng. committed).

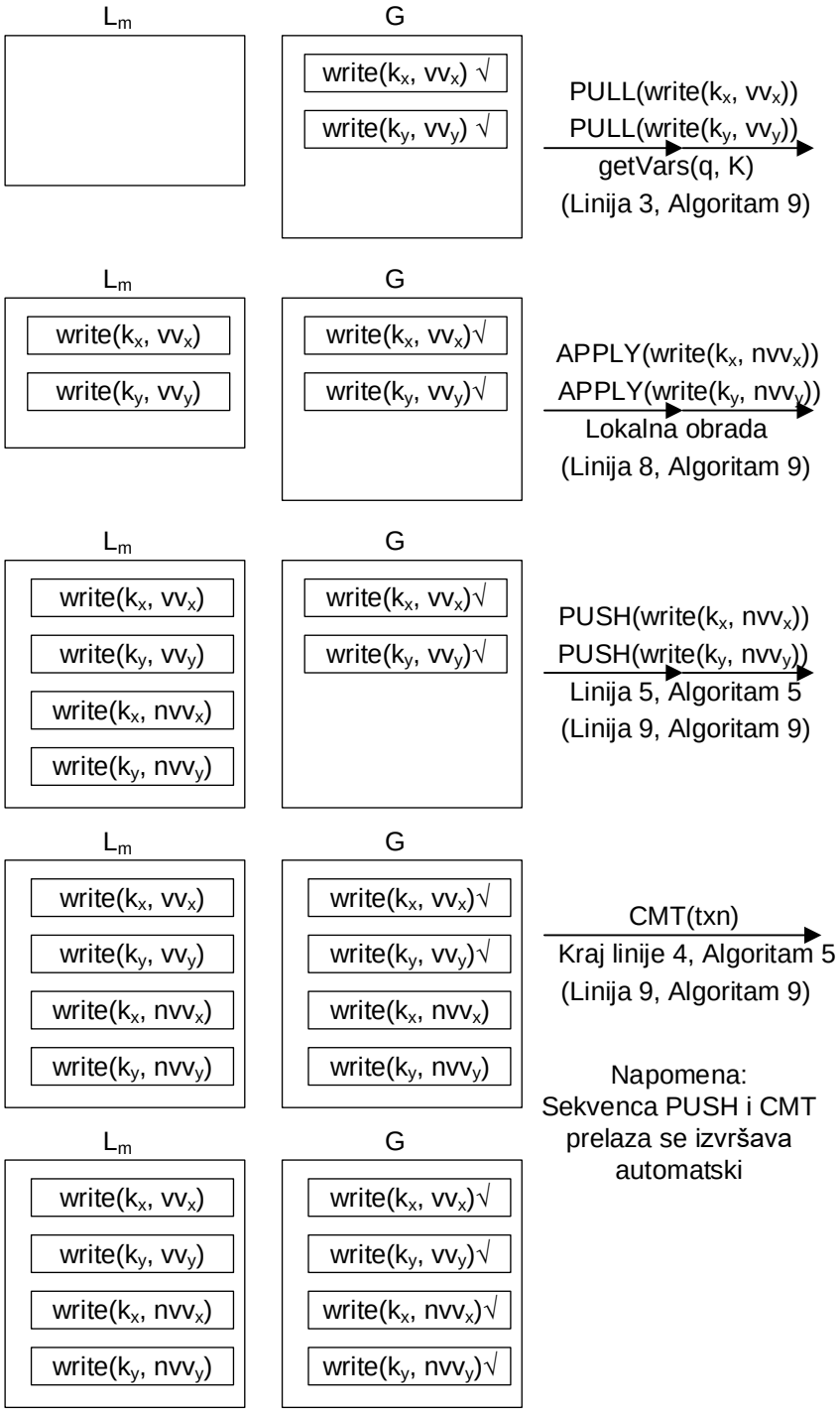
Algoritam 9 – Instanca algoritma T za prenos novca, Tm

```
1: Tm(K, p)    // pretpostavimo da je K = [ kx, ky ]
2:   kx := K[0], ky := K[1]
3:   V := dpstm.getVars(K )
4:   (ex, vvx) := V[0]
5:   (ey, vvy) := V[1]
6:   nvx := vvx.val – p
7:   nvy := vvy.val + p
8:   W := R := [ (kx, vvx.ver, nvx), (ky, vvy.ver, nvy) ]
9:   v := dpstm.commitVars([R, W] )
```

Promenljive korišćene u tabeli Algoritam 9 i na Slici 8 su sledeće. Ulazni parametri od T_m su: *K* je lista ključeva za promenlive *x* i *y*, K = [k_x, k_y], a *p* je iznos novca za prenos sa *x* na *y*. Promenljive *V*, (*e*, *vv*), *R* i *W* su objašnjene u odeljku 4.2. Podsedimo se da je *vv* = (*ver*, *val*). Polja *vv.ver* i *vv.val* su jednaka elementima para *vv* označenim sa *ver* i *val*, respektivno. Tako su $vv_x.val$ i $vv_y.val$ ulazne vrednosti t-promenljivih *x* i *y*, respektivno. Promenljive nv_x i nv_y su izlazne (tj. nove) vrednosti (*val*) za *x* i *y*, respektivno.

Linije 1-3 u Algoritmu 9 odgovaraju prvom redu na Slici 8. U ovoj sekciji Algoritma 9, transakcija T_m poziva getVars(*K*), vidi liniju 3 u Algoritmu 9. Ovaj poziv se preslikava na dva pravila PULL (ova dva pravila su prikazana kao kaskada dve strelice na Slici 8), što dovodi do povlačenja operacija write(k_x, vv_x) i write (k_y, vv_y), gde su vv_x i vv_y parovi koji sadrže (ver_x, val_x) i (ver_y, val_y), koji sadrže verzije i vrednosti t-promenljivih *x* i *y*, respektivno. Primetimo da se ove dve operacije nakon povlačenja pojavljuju u lokalnom logu L_i u drugom redu na Slici 8.

Linije 4-7 u Algoritmu 9 nemaju odgovarajući deo na Slici 8. U ovoj sekciji Algoritma 9, T_m obavlja svoju lokalnu obradu kako bi odredila izlazne vrednosti t-promenljivih *x* i *y* (nv_x i nv_y, respektivno).



Slika 8 – Izvršenje Algoritma 9 u okviru push/pull semantičkog modela

Linija 8 u Algoritmu 9 odgovara drugom redu na Slici 8. U ovoj sekciji Algoritma 9, T_m ažurira svoje lokalne kopije t-promenljivih *x* i *y*. Ona to radi formiranjem lista *R* i *W*, koji će koristiti funkcija commitVars u liniji 9 u Algoritmu 9.

Kao rezultat ovih ažuriranja, operacije write(k_x, nvv_x) i write(k_y, nvv_y) se dodaju na kraj lokalnog loga transakcije T_m, vidi treći red na Slici 8.

Linija 9 u Algoritmu 9 odgovara trećem, četvrtom i petom redu na Slici 8. U ovoj sekciji Algoritma 9, T_m poziva commitVars(C), a ovaj poziv se preslikava u dva odgovarajuća pravila PUSH i CMT (dva pravila PUSH koji su prikazani kao kaskada dve strelice na Slici 8 i jedno pravilo CMT), koja se izvršavaju atomično, zato što se ceo Algoritam 5 izvršava atomično. Kao rezultat dva pravila PUSH, dve nezavršene write operacije su dodate na kraj globalnog loga G, vidi G u četvrtom redu na Slici 8, primetimo da te dve nove operacije nisu označene (eng. checked). Na kraju, kao rezultat pravila CMT, poslednje dve operacije se uspešno završavaju (eng. committed), vidi G u petom redu na Slici 8.

4.5 Dokazi kriterijuma korektnosti (D)PSTM

U ovom odeljku se dokazuje da se (D)PSTM mogu serijalizovati, dokazivanjem da push/pull semantički model (D)PSTM (Definicija 2) zadovoljava preduslove za odgovarajuća push/pull pravila. U sledećim Lemama 1 do 4, se za svaki preduslov, u prvoj rečenici podseća (iz [37]) šta taj preduslov zahteva, a zatim se u drugoj rečenici dokazuje da je taj zahtev zadovoljen.

Lema 1: Push/Pull model (D)PSTM zadovoljava preduslove za pravilo APPLY.

Dokaz. Slede zahtevi i dokazi za svaki preduslov.

Preduslov (i) $c_1 \downarrow_{\mathrm{tx}} (m, c_2)$ zahteva postojanje putanje u programskom kodu c_1 do poziva operacije m i programskog koda c_2 za nastavak (posle poziva m). Ovaj zahtev je zadovoljen po definiciji transakcionog algoritma T, jer postoji putanja u T do write operacije m.

Preduslov (ii) L_1 allows $\langle m, \sigma_1, \sigma_2 \rangle$ zahteva da je operacija m dozvoljena od strane lokalnog loga L_1. Ovaj zahtev je zadovoljen po definiciji sekvencijalne specifikacije (Definicija 3) lokalnog loga L_1 i write operacije m.

Preduslov (iii) $\mathrm{fresh}(id)$ zahteva da je na L_1 dodata nova operacija sa jedinstvenom identifikacijom id. Ovaj zahtev je zadovoljen po konstrukciji L_1. Q.E.D.

Lema 2: Push/Pull model (D)PSTM zadovoljava preduslove za pravilo PUSH.

Dokaz. Slede zahtevi i dokazi za svaki preduslov.

Preduslov (*i*) $op \lhd \lfloor L_1 \rfloor_{\text{unpushed}}$ zahteva da se operacija *op* može prebaciti s leve strane svih operacija u L_1 sa status flegom unpushed (tj. one koje nisu gurnute u globalni log). Ovaj zahtev je zadovoljen zato što su t-promenljive u *W* jedinstvene (*W* je skup predstavljen kao lista), pa se odgovarajuće write operacije mogu gurati u globalni log u bilo kom redosledu (ova argumentacija iz [28] je tačna, dok je u [32] načinjen lapsus i ta argumentacija je netačna).

Preduslov (*ii*) $\lfloor G_1 \rfloor_{\text{gUCmt}} \backslash \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \lhd op$ zahteva da sve operacije u G_1 sa status flegom gUCmt (tj. one koje nisu završene), bez operacija koje je gurnula tekuća transakcija, mogu prebaciti s leve strane (autori u [37] su načinili lapsus i napisali „s desne strane“) tekuće operacije *op*. Ovaj zahtev je ispunjen jer po push/pull modelu (D)PSTM (Definicija 2), nema nezavršenih (eng. uncommitted) operacija u G_1 (bez onih operacija koje je gurnula tekuća transakcija).

Preduslov (*iii*) G_1 allows *op* zahteva da je operacija *op* dozvoljena od stane loga G_1. Ovaj zahtev je zadovoljen po definciji sekvencijalne specifikacije (Definicija 3) loga G_1 i write operacije *op*. Q.E.D.

Lema 3: Push/Pull model (D)PSTM zadovoljava preduslove za pravilo PULL.

Dokaz. Slede zahtevi i dokazi za svaki preduslov.

Preduslov (*i*) $op \notin L$ zahteva da operacija *op* nije već ranije povučena. Ovaj zahtev je zadovoljen po definiciji generičkog transakcionog algoritma nad (D)PSTM T, koja zahteva da se svaka t-promenljiva povlači samo jedanput.

Preduslov (*ii*) *L* allows *op* zahteva da je operacija *op* dozvoljena od strane loga *L*. Ovaj zahtev je zadovoljen po definciji sekvencijalne specifikacije (Definicija 3) loga *L* i write operacije *op*.

Preduslov (*iii*) $op \lhd \lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}$ zahteva da se operacija *op* može prebaciti s leve strane svih operacija koje je primenila tekuća transakcija. Ovaj zahtev je zadovoljen po definiciji generičkog transakcionog algoritma nad (D)PSTM T, koji prvo povlači operacije iz globalnog loga (pomoću funkcije getVars), pa tek posle toga primenjuje lokalne operacije (pripremajući listu *W* za završni poziv funkcije commitVars), tako da u trenutku kada se povlači operacija *op*, skup primenjenih operacija ($\lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}$) je prazan, a svaka operacija se može prebaciti s leve strane praznog skupa (ovaj dokaz je upotpunjen u odnosu na [28]). Q.E.D.

Lema 4: Push/Pull model (D)PSTM zadovoljava preduslove za pravilo CMT.

Dokaz. Slede zahtevi i dokazi za svaki preduslov.

Preduslov (i) $\mathrm{fin}(c)$ zahteva postojanje putanje kroz programski kod tx *c*, koja vodi do završnog iskaza skip. Ovaj zahtev je zadovoljen po definiciji generičkog transakcionog algoritma nad (D)PSTM T, zato što je on pravolinijski i uvek se završava.

Preduslov (ii) $L_1 \subseteq G_1$ zahteva da lokalni log L_1 transakcije mora biti podskup globalnog loga G_1. Ovaj zahtev je zadovoljen po definiciji generičkog transakcionog algoritma nad (D)PSTM T i po konstrukciji push/pull semantičkog modela (D)PSTM (Definicija 2).

Preduslov (iii) $\forall op \in \lfloor L_1 \rfloor_{\mathrm{pulled}}, (op, g) \in G_1.\, g = gCmt$ zahteva da su sve operacije koje je povukla tekuća transakcija, bile gurnute od strane transakcija koje su se u međuvremenu uspešno završile (eng. committed). Ovaj zahtev je zadovoljen jer G_1, u slučaju (D)PSTM, sadrži samo uspešno završene operacije (vidi Definiciju 2).

Preduslov (iv) $\mathrm{cmt}(G_1, L_1, G_2)$ zahteva da je tekući globalni log G_1 ažuriran na naredni log G_2 (pomoću predikata cmt) označavanjem operacija gurnutih od strane tekuće transakcije kao uspešno završenih (eng. committed). Ovaj zahtev je zadovoljen preslikavanjem poziva commitVars – u slučaju uspešnog završetka, sve write operacije koje su specificirane sa W se označavaju kao uspešno završene (Definicija 2 i Definicija 2.3). Q.E.D.

Teorema 1 ispod direktno sledi iz Lema 1 do 4, jer push/pull semantički model (D)PSTM zadovoljava preduslove relevantnih push/pull pravila.

Teorema 1 (Mogućnost serijalizacije (D)PSTM): (D)PSTM se može serijalizovati.

5. ZAKLJUČAK

U ovoj doktorskoj disertaciji je pretstavljena jedna formalno verifikovana distribuirana softverska transakciona memorija otporna na otkaze, koja je deterministička i implementirana kao prirodno proširenje postojećih apstrakcija programskog jezika Pajton, i čija formalna verifikacija je urađena na bazi njenog push/pull semantičkog modela. U narednim odeljcima su izloženi doprinosi i mogućnost primene, prednosti i nedostaci, i pravci budućeg rada.

5.1 Doprinosi i mogućnost primene

Osnovni ciljevi istraživanja obuhvaćenog disertacijom su uspešno realizovani: (1) razvijena je DPSTM v3 kao sledeći prirodan korak u DPSTM evoluciji, u kom se jedan TC zamenjuje parom TC koji rade u režimu vodeći-prateći, kako bi se obezbedila otpornost na otkaze pojedinačnih TC, i (2) formalno je verifikovana DPSTM v3 na osnovu njenog push/pull semantičkog modela. Konstruisana i verifikovana na ovaj način, DPSTM v3 pretstavlja jedno rešenje predmeta (problema) ove doktorske teze. Na osnovu istraživanja iznetog u disertaciji i dostupne literature, može se konstatovati da je DPSTM v3 prva formalno verifikovana distribuirana softverska transakciona memorija otporna na otkaze, koja je deterministička i implementirana kao prirodno proširenje postojećih apstrakcija programskog jezika Pajton.

Glavni rezultati, odnosno doprinosi, ove disertacije su: (1) distribuirani automat sa konačnim brojem stanja za upravljanje parom transakcionih koordinatora u režimu vodeći-prateći, (2) adaptirtani deterministički protokol za replikaciju podataka, (3) procedura za oporavak servisa nakon otkaza koordinatora, (4) odgovarajući push/pull semantički model i (5) formalni dokazi da su zadovoljeni kriterijumi korektnosti.

Pretstavljeni rezultati se mogu koristiti kako za akademska, tako i u polju industrijskih istraživanja. Rezultati se mogu posmatrati u dva aspekta. Sa jedne strane, demonstrirana je primenljivost pristupa formalnoj verifikaciji distribuiranih softverskih transakcionih memorija upotrebom push/pull semantičkog modela na primeru konkretne DSTM, a konkretan model i dokazi mogu poslužiti kao mustre za formalnu verifikaciju drugih DSTM. S druge strane, razvijeno je rešenje DSTM za programski jezik Pajton, koje je otporno na otkaze, i čija eksperimentalna evaluacija u laboratoriji je pokazala da ima zadovoljavajuće performanse. S obzirom da je u današnje vreme Pajton prvi na IEEE listi najpopularnijih programskih jezika [48], potencijal primenljivosti ovakve komponente je veoma visok.

Primarni aplikacioni domen razvijene DPSTM v3 su svakako inteligentni ugrađeni sistemi zasnovani na IoT u ivičnim mrežama Interenta, kao što su pametne kuće, automobili, itd. Međutim, šire gledano DPSTM v3 se može koristiti u širokom opsegu aplikacionih domena, polazeći od velikih simulacionih programskih paketa kao što je DEEPSAM paket za farmaceutsku industriju, preko kritičnih infrastruktura kao što su sajber-fizički sistemi i Internet stvari, i dalje preko mrežnih oblaka, do sistema za podršku nauke o podacima i unapređenog obrazovanja.

5.2 Prednost i nedostaci

U ovom odeljku se razmatraju prednosti i nedostaci iz perspektive eksperimentalne evaluacije, arhitekture sistema, i formalne verifikacije.

Iz perspektive eksperimentalne evaluacije, rezultati su generalno pozitivni, i konkretno pokazuju sledeće: (1) da se propusnost sistema povećava super-linearno kako se udeo read operacija povećava od 0% do 100% (a udeo write operacija simetrično smanjuje), i (2) da je propusnost sistema za mrežnu konfiguraciju sa kolokacijom mnogo bolja od propusnosti sistema za potpuno distribuiranu mrežnu konfiguraciju.

Prednosti primenjenog pristupa eksperimentalnoj evaluaciji su sledeće: (1) ovaj pristup obezbeđuje dobro pokrivanje celog opsega mogućih aplikacija, (2) ovaj pristup omogućava dobijanje opštih trendova, kao i donje i gornje granice performanse, i (3) ovaj pristup omogućava estimaciju performanse (tj. propusnosti sistema) za proizvoljno radno opterećenje korišćenjem interpolacije.

Ograničenja konkretne eksperimentalne evaluacije DPSTM v3 prikazane u ovom radu su sledeće: (1) ona pokriva samo dve mrežne konfiguracije, i (2) ona je urađena za jednu konkretnu eksperimentalnu postavku korišćenjem jedne vrste hardvera i jedne mrežne tehnologije.

Iz perspektive arhitekture, glavne prednosti DPSTM v3 su što je otorna na otkaze, deterministička, i izgrađena na postojećim arhitektonskim apstrakcijama jezika Pajton, konkretno iz paketa „multiprocessing“. Jedno ograničenje DPSTM v3 je da ne podržava automatsko uravnoteženje opterećenja lokalnih DS servera. Šire gledano, DPSTM v3 je jedno rešenje predmeta ove disertacije, međutim moguće je da postoje i druga rešenja, i to pitanje definitivno ostaje otvoreno za buduća istraživanja.

Iz perspektive formalne verifikacije, prednosti pristupa zasnovanog na push/pull semantičkom modelu su: (1) postoje realna očekivanja da će taj pristup biti šire korišćen kao faktički standard u ovoj oblasti, i (2) prilično je jednostavan i intuitivan za korišćenje. Ograničenja tog pristupa su: (1) za generički push/pull semantički model jeste dokazano svojstvo mogućnost serijalizacije ali nije dokazano svojstvo životnosti, i (2) formalni dokazi u originalnom radu o push/pull modelu, pa posledično i u ovoj disertaciji, su urađeni ručno. Ta pitanja takođe ostaju otvorena za buduća istraživanja.

5.3 Pravci budućeg rada

U cilju prevazilaženja uočenih ograničenja, planirano je da se u budućem radu: (1) nastavi eksperimentalna evaluacija na drugim mrežnim konfiguracijama, kao i korišćenjem drugih vrsta hardvera i drugih mrežnih tehnologija, (2) nastavi istraživanje drugih mogućih DPSTM arhitektura, (3) nastavi istraživanje mogućnosti dokazivanja svojstva životnosti, i (4) nastavi istraživanje sa ciljem korišćenja nekih programskih alata, kao što su npr. Maude i Ott, radi pravljenja izvršivih push/pull semantičkih modela i automatskog dokazivanja željenih svojstava, čime bi se dobio još viši nivo ubeđenja u dobijene rezultate.

Pored ovih aktivnosti na prevazilaženju postojećih ograničenja, u budućem radu je planirano da se istraži mogućnost primene DPSTM v3 u: (1) rešenju pametne kuće zasnovane na Internetu stvari i računarskom oblaku, i (2) u računarsko-hemijskim simulacijama sa programskim paketom DEEPSAM na klasteru računara.

LITERATURA

[1] M. Herlihy, J.E.B. Moss, "Transactional memory: architectural support for lockfree data structures," in Proc. 20th Annual International Symposium on Computer Architecture, pp. 289–300, 1993. doi: 10.1145/165123.165164

[2] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in Proc. of the 12th International Symposium on Computer Architecture, pp. 124–131, 1983. doi: 10.1145/800046.801647

[3] N. Shavit, D. Touitou, "Software transactional memory," in Proc. 14th ACM Symposium on Principles of Distributed Computing, pp. 204–213, 1995. doi: 10.1145/224964.224987

[4] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster, " in Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 198–208, 2006. doi: 10.1145/1122971.1123002

[5] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C.C. Kirkham, I. Watson, "DiSTM: A software transactional memory framework for clusters," in Proc. 37th International Conference on Parallel Processing, pp. 51-58, 2008. doi: 10.1109/ICPP.2008.59

[6] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C.C. Kirkham, I. Watson, "Investigating software transactional memory on clusters," in Proc. 22nd International Parallel and Distributed Processing Symposium, pp. 1-6, 2008. doi: 10.1109/IPDPS.2008.4536340

[7] M. Couceiro, P. Romano, N. Carvalho, L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in Proc. 15th Pacific Rim International Symposium on Dependable Computing, pp. 307-313, 2009. doi: 10.1109/PRDC.2009.55

[8] J. Cachopo, A. Rito-Silva, "Versioned boxes as the basis for memory transactions," Science of Computer Programming, vol. 63, no. 2, pp. 172-185,

2006. doi: 10.1016/j.scico.2006.05.009

[9] J. Cachopo, A. Rito-Silva, “Combining software transactional memory with a domain modeling language to simplify web application development,” in Proc. 6th International Conference on Web Engineering, pp. 297–304, 2006. doi: 10.1145/1145581.1145640

[10] P. Romano, N. Carvalho, L. Rodrigues, “Towards distributed software transactional memory systems,” in Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware, pp. 1-4, 2008. doi: 10.1145/1529974.1529980

[11] N. Carvalho, J. Cachopo, L. Rodrigues, A. Rito-Silva, “Versioned transactional shared memory for the FenixEDU web application,” in Proc. 2nd Workshop on Dependable Distributed Data Management, pp. 15–18, 2008. doi: 10.1145/1435523.1435526

[12] R. L. Bocchino, V. S. Adve, B. L. Chamberlain, “Software transactional memory for large scale clusters,” in Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 247–258, 2008. doi: 10.1145/1345206.1345242

[13] J. Kim, B. Ravindran, “Scheduling transactions in replicated distributed software transactional memory,” in Proc. 13th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing, pp. 227–234, 2013. doi: 10.1109/CCGrid.2013.88

[14] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, L. Rodrigues, “On speculative replication of transactional systems,” Journal of Computer and System Sciences, vol. 80, no. 1, pp. 257–276, 2014. doi: 10.1016/j.jcss.2013.07.006

[15] M. M. Saad, B. Ravindran, “Snake: Control flow distributed software transactional memory,” in Proc. Stabilization, Safety, and Security of Distributed Systems, pp. 238–252, 2011. doi: 10.1007/978-3-642-24550-3_19

[16] M. Herlihy, Y. Sun, “Distributed transactional memory for metric-space networks,” Distributed Computing, vol. 20, no. 3, pp. 195–208, 2007. doi: 10.1007/s00446-007-0037-x

[17] G. Sharma and C. Busch. Distributed transactional memory for general networks. Distributed Computing, vol. 27, no. 5, pp. 329–362, 2014.

doi:10.1007/s00446-014-0214-7

[18] B. Zhang, B. Ravindran, R. Palmieri, “Distributed transactional contention management as the traveling salesman problem,” in Proc. Structural Information and Communication Complexity, pp. 54–67, 2014. doi: 10.1007/978-3-319-09620-9_6

[19] C. Busch, M. Herlihy, M. Popovic, G. Sharma, “Fast scheduling in distributed transactional memory,” in Proc. Symposium on Parallelism in Algorithms and Architectures, pp. 173–182, 2017. doi: 10.1145/3087556.3087565

[20] C. Busch, M. Herlihy, M. Popovic, G. Sharma, “Time-communication impossibility results for distributed transactional memory,” Distributed Computing, vol. 31, no. 6, pp. 471–487, 2018. doi:10.1007/s00446-017-0318-y

[21] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, A. Suissa, “Exploiting locality in lease-based replicated transactional memory via task migration,” in Proc. International Symposium on Distributed Computing, pp. 121–133, 2013. doi: 10.1007/978-3-642-41527-2_9

[22] R. Palmieri, S. Peluso, B. Ravindran, “Transaction execution models in partially replicated transactional memory: The case for data-flow and control-flow,” in: R. Guerraoui, P. Romano (Eds.) Transactional Memory – Foundations, Algorithms, Tools, and Applications, LNCS, vol. 8913, Springer, pp. 341–366. 2015. doi: 10.1007/978-3-319-14720-8_16

[23] S. Bagchi, M.-B. Siddiqui, P. Wood, H. Zhang, “Dependability in Edge Computing,” Communications of ACM, vol. 63, no. 1, pp. 59-66, 2020. doi:10.1145/3362068

[24] K. Gilly, S. Filiposka, A. Mishev “Supporting Location Transparent Services in a Mobile Edge Computing Environment,” Advances in Electrical and Computer Engineering, vol. 18, no. 4, pp. 11-22, 2018. doi: 10.4316/AECE.2018.04002

[25] M. Popovic, B. Kordic, “PSTM: Python Software Transactional Memory,” in Proc. 22nd IEEE Telecommunications Forum, pp. 1106-1109, 2014. doi: 10.1109/TELFOR.2014.7034600

[26] A. Liu, H. Zhu, M. Popovic, S. Xiang, L. Zhang, “Formal Analysis and Verification of the PSTM Architecture Using CSP”, The Journal of Systems and Software, vol. 165, 2020. doi: 10.1016/j.jss.2020.110559

[27] B. Kordic, M. Popovic, S. Ghilezan, “Formal Verification of Python Software Transactional Memory Based on Timed Automata”, Acta Polytechnica Hungarica, Journal of Applied Sciences, vol. 16, no. 7, pp. 197-216, 2019. doi: 10.12700/APH.16.7.2019.7.12

[28] M. Popovic, M. Popovic, S. Ghilezan, B. Kordic, “Formal Verification of Python Software Transactional Memory Serializability Based on the Push/Pull Semantic Model,” in Proc. 6th Conference on the Engineering of Computer Based Systems, Article No. 6, pp. 1-8, 2019. doi: 10.1145/3352700.3352706

[29] M. Popovic, B. Kordic, M. Popovic, I. Basicevic, “Online Algorithms for Scheduling Transactions on Python Software Transactional Memory,” Serbian Journal of Electrical Engineering, vol. 16, no. 1, pp. 85-104, 2019. doi:10.2298/SJEE1901085P

[30] C. Xu, X. Wu, H. Zhu, M. Popovic, „Modeling and Verifying Transaction Scheduling for Software Transactional Memory using CSP,“ in Proc. 13th Theoretical Aspects of Software Engineering Symposium, pp. 240-247, 2019. doi: 10.1109/TASE.2019.00009

[31] M. Popovic, B. Kordic, M. Popovic, I. Basicevic, “A Solution of Concurrent Queue on Local and Distributed Python STM,” Telfor Journal, vol. 11, no. 1, pp. 64-69, 2019. doi:10.5937/telfor1901064P

[32] M. Popovic, M. Popovic, S. Ghilezan, B. Kordic, “Formal Verification of Local and Distributed Python Software Transactional Memeory,” Revue Roumaine des Sciences Techniques. Ser. Electrotechnique et Energetique, vol. 64, no. 4, pp. 423–428, 2019.

[33] M. Popovic, M. Popovic, B. Kordic, I. Basicevic, “A Solution of Python Distributed STM Based on Data Replication,” in Proc. 27th IEEE Telecommunications Forum, pp. 1-4, 2019. doi: 10.1109/TELFOR48224.2019.8971069

[34] M. van Steen, A.S. Tanenbaum, Distributed Systems, 3rd edition, Published by Maarten van Steen, pp. 355-421, 2017.

[35] W. Vogels, “Eventually consistent,” Communications of the ACM, vol. 52, no. 1, pp. 40–44, 2009. doi: 10.1145/1435417.1435432

[36] D.J. Abadi, J.M. Faleiro, “An Overview of Deterministic Database Systems,”

Communications of the ACM, vol. 81, no. 9, pp. 78–88, 2018. doi: 10.1145/3181853

[37] E. Koskinen, M. Parkinson, “The Push/Pull Model of Transactions,” in Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 186-195, 2015. doi: 10.1145/2737924.2737995

[38] D. Dimitrov, V. Raychev, M. Vechev, E. Koskinen, “Commutativity Race Detection,” in Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 305-315, 2014. doi: 10.1145/2594291.2594322

[39] T.D. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, “Proust, A Design Space for Highly-Concurrent Transactional Data Structures,” Cornell University Library, arXiv:1702.04866v2 [cs.DC] 26 Jun 2017, pp. 1-20, 2017.

[40] E. Nan, U. Radosavac, M. Matić, I. Stefanović, I. Papp and M. Antić, "One solution for voice enabled smart home automation system," in Proc. IEEE 7th International Conference on Consumer Electronics - Berlin, pp. 132-133, 2017. doi: 10.1109/ICCE-Berlin.2017.8210611

[41] E. Nan, U. Radosavac, I. Papp and M. Antić, "Architecture of voice control module for smart home automation cloud," in Proc. IEEE 7th International Conference on Consumer Electronics - Berlin, pp. 97-98, 2017. doi: 10.1109/ICCE-Berlin.2017.8210601

[42] B. Kordic, M. Popovic, M. Popovic, M. Goldstein, M. Amitay, D. Dayan, “A Protein Structure Prediction Program Architecture Based on a Software Transactional Memory,” in Proc. 6th Conference on the Engineering of Computer Based Systems, Article No. 1, pp. 1-9, 2019. doi: 10.1145/3352700.3352701

[43] P. Hunt, M. Konar, F. P. Junqueira, B. Reed, “ZooKeeper: wait-free coordination for internet-scale systems,” in Proc. of the 2010 USENIX conference on USENIX annual technical conference, pp. 145-158, 2010.

[44] M. Popovic, I. Basicevic, M. Djukic, M. Popovic, “Fault Tolerant Distributed Python Software Transactional Memory,” Advances in Electrical and Computer Engineering, vol. 20, no. 4, pp. 19-28, 2020. doi: 10.4316/AECE.2020.04003

[45] L. Martinovic, D. Capko, A. Erdeljan, “Load Balancing of Large Distribution

Network Model Calculations," Advances in Electrical and Computer Engineering, vol. 17, no. 4, pp. 11-18, 2017. doi: 10.4316/AECE.2017.04002

[46] S. Gilbert, N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM SIGACT News, vol. 33, no. 2, pp. 51-59, 2002. doi: 10.1145/564585.564601

[47] E. Koskinen, M. Parkinson, "The push/pull model of transactions (extended version)". Tech. Rep. RC25529, IBM Research, 2015. https://cs.nyu.edu/media/publications/TR2014-967.pdf

[48] "The top programming languages: Our latest rankings put Python on top–again," IEEE Spectrum, August 2020, pp. 22, 2020.

Овај Образац чини саставни део докторске дисертације, односно докторског уметничког пројекта који се брани на Универзитету у Новом Саду. Попуњен Образац укоричити иза текста докторске дисертације, односно докторског уметничког пројекта.

План третмана података

Назив пројекта/истраживања
Формално верификована дистрибуирана софтверска трансакциона меморија отпорна на отказе
Назив институције/институција у оквиру којих се спроводи истраживање
а) Универзитет у Новом Саду, Факултет техничких наука б) в)
Назив програма у оквиру ког се реализује истраживање
1. Опис података
1.1 Врста студије *Укратко описати тип студије у оквиру које се подаци прикупљају* **У овој студији нису прикупљани подаци.** _____ _____ 1.2 Врсте података а) квантитативни б) квалитативни 1.3. Начин прикупљања података а) анкете, упитници, тестови

б) клиничке процене, медицински записи, електронски здравствени записи

в) генотипови: навести врсту _____

г) административни подаци: навести врсту _____

д) узорци ткива: навести врсту_____

ђ) снимци, фотографије: навести врсту_____

е) текст, навести врсту _____

ж) мапа, навести врсту _____

з) остало: описати _____

1.3 Формат података, употребљене скале, количина података

1.3.1 Употребљени софтвер и формат датотеке:

a) Excel фајл, датотека _____

b) SPSS фајл, датотека _____

c) PDF фајл, датотека _____

d) Текст фајл, датотека _____

e) JPG фајл, датотека _____

f) Остало, датотека _____

1.3.2. Број записа (код квантитативних података)

а) број варијабли _____

б) број мерења (испитаника, процена, снимака и сл.) _____

1.3.3. Поновљена мерења

а) да

б) не

Уколико је одговор да, одговорити на следећа питања:

а) временски размак између поновљених мера је _____

б) варијабле које се више пута мере односе се на _____

в) нове верзије фајлова који садрже поновљена мерења су именоване као _____

Напомене: _____

Да ли формати и софтвер омогућавају дељење и дугорочну валидност података?

а) Да

б) Не

Ако је одговор не, образложити _____

_____

2. Прикупљање података

2.1 Методологија за прикупљање/генерисање података

2.1.1. У оквиру ког истраживачког нацрта су подаци прикупљени?

а) експеримент, навести тип _____

б) корелационо истраживање, навести тип _____

ц) анализа текста, навести тип _____

д) остало, навести шта _____

2.1.2 Навести врсте мерних инструмената или стандарде података специфичних за одређену научну дисциплину (ако постоје).

_____

_____

2.2 Квалитет података и стандарди

2.2.1. Третман недостајућих података

а) Да ли матрица садржи недостајуће податке? Да Не

Ако је одговор да, одговорити на следећа питања:

а) Колики је број недостајућих података? _____

б) Да ли се кориснику матрице препоручује замена недостајућих података? Да Не

в) Ако је одговор да, навести сугестије за третман замене недостајућих података

_____

2.2.2. На који начин је контролисан квалитет података? Описати

_____

_____

2.2.3. На који начин је извршена контрола уноса података у матрицу?

_____

_____

3. Третман података и пратећа документација

3.1. Третман и чување података

3.1.1. Подаци ће бити депоновани у _____ репозиторијум.

3.1.2. URL адреса _____

3.1.3. DOI _____

3.1.4. Да ли ће подаци бити у отвореном приступу?

а) Да

б) Да, али после ембарга који ће трајати до _____

в) Не

Ако је одговор не, навести разлог _____

3.1.5. Подаци неће бити депоновани у репозиторијум, али ће бити чувани.

Образложење

_____

_____

3.2 Метаподаци и документација података

3.2.1. Који стандард за метаподатке ће бити примењен? _____

3.2.1. Навести метаподатке на основу којих су подаци депоновани у репозиторијум.

_____

_____

Ако је потребно, навести методе које се користе за преузимање података, аналитичке и процедуралне информације, њихово кодирање, детаљне описе варијабли, записа итд.

_____

_____

_____

_____

3.3 Стратегија и стандарди за чување података

3.3.1. До ког периода ће подаци бити чувани у репозиторијуму? _____

3.3.2. Да ли ће подаци бити депоновани под шифром? Да Не

3.3.3. Да ли ће шифра бити доступна одређеном кругу истраживача? Да Не

3.3.4. Да ли се подаци морају уклонити из отвореног приступа после извесног времена?

Да Не

Образложити

_____

_____

4. Безбедност података и заштита поверљивих информација

Овај одељак МОРА бити попуњен ако ваши подаци укључују личне податке који се односе на учеснике у истраживању. За друга истраживања треба такође размотрити заштиту и сигурност

података.

4.1 Формални стандарди за сигурност информација/података

Истраживачи који спроводе испитивања с људима морају да се придржавају Закона о заштити података о личности *(https://www.paragraf.rs/propisi/zakon_o_zastiti_podataka_o_licnosti.html)* и одговарајућег институционалног кодекса о академском интегритету.

4.1.2. Да ли је истраживање одобрено од стране етичке комисије? Да Не

Ако је одговор Да, навести датум и назив етичке комисије која је одобрила истраживање

_____

4.1.2. Да ли подаци укључују личне податке учесника у истраживању? Да Не

Ако је одговор да, наведите на који начин сте осигурали поверљивост и сигурност информација везаних за испитанике:

а) Подаци нису у отвореном приступу

б) Подаци су анонимизирани

ц) Остало, навести шта

_____

_____

5. Доступност података

5.1. Подаци ће бити

а) јавно доступни

б) доступни само уском кругу истраживача у одређеној научној области

ц) затворени

Ако су подаци доступни само уском кругу истраживача, навести под којим условима могу да их користе:

_____

_____

Ако су подаци доступни само уском кругу истраживача, навести на који начин могу приступити подацима:

_____

_____

5.4. Навести лиценцу под којом ће прикупљени подаци бити архивирани.

_____

6. Улоге и одговорност

6.1. Навести име и презиме и мејл адресу власника (аутора) података

_____

6.2. Навести име и презиме и мејл адресу особе која одржава матрицу с подацима

_____

6.3. Навести име и презиме и мејл адресу особе која омогућује приступ подацима другим истраживачима

_____