Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.41
DOI: https://doi.org/10.24867/02BE30Jelaca

PRIMENA AKTER PROGRAMSKOG MODELA NA MIKROSERVISNU ARHITEKTURU SERVISA ZA SKLADIŠTENJE PODATAKA U MEMORIJI

APPLICATION OF ACTOR PROGRAMMING MODEL ON MICROSERVICE ARCHITECTURE FOR IN-MEMORY DATA STORAGE SERVICE

Marko Jelača, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu analizirano je skladištenje podataka u memoriji primenom akter programskog modela. Predložena je mikroservisna arhitektura servisa za skladištenje podataka u memoriji upotrebom Microsoft Azure Service Fabric platforme. Predloženo rešenje se sastoji od menadžer aktera koji upravljaju velikim brojem aktera koji skladište podatke u svojoj memoriji. Efikasna raspodela aktera po particijama je realizovana upotrebom algoritma za particionisanje brojeva. Dostupnost i pouzdanost su obezbeđeni replikacijom i trajnim čuvanjem podataka u bazi podataka. Performanse rešenja su merene na lokalnom klasteru promenom broja aktera.*

Ključne reči: *Skladištenje podataka u memoriji, akter programski model, mikroservisna arhitektura.*

Abstract – *This paper analyzes data storing in memory using actor programming model. Paper proposes a microservice architecture for in-memory data storage service using Microsoft Azure Service Fabric platform. The proposed solution consist of manager actors who coordinates with large number of actors for data storing. Evenly distribution of actors across partitionis is done using number partitioning algorithm. Availability and reliability are provided with replication and permanent data storage in the database. Performance of the proposed solution was measured on local cluster, changing the number of actors.*

Keywords: *Storing in memory, akter programming model, microservice architecture*

1. UVOD

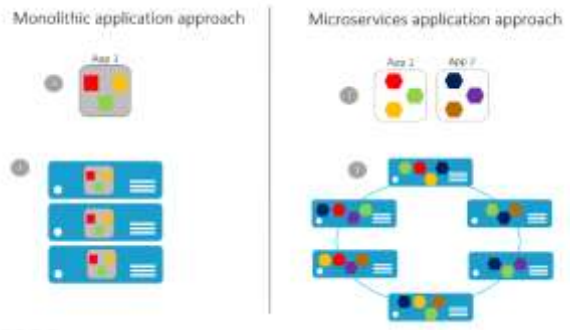
Skladištenje podataka u memoriji je jedan od osnovih koncepata u računarstvu. Osnovni princip jeste da skladištenje podataka u memoriji ubrzava operacije. Kada želimo da dobavimo neki podatak, čije dobavljanje je relativno skupa operacija po pitanju vremena i resursa, mi taj podataka skladištimo u memoriji kako bi sledeće dobavljanje tog podatka bilo jeftinije. Neki od glavnih razloga zašto je skladištenje podataka u velikim sistemima veoma važno jeste to što je skladištenje podataka najvažniji deo softvera koji utiče na to koliko brzo aplikacija reaguje na zahteve, kao i činjenica da gubitak podataka nije prihvatljiv u poslovnim operacijama.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Srđan Vukmirović, vanr.prof.

Računarstvo u oblaku (eng. *cloud computing*) predstavlja model koji omogućava pristup na zahtev deljenim računarskim resursima (kao što su računarska mreža, serveri, skladište podataka i servisi) [1]. Neke od karakteristika ovog modela jesu korišćenje neograničenih resursa, smanjenje infrastrukturnih rizika kupovinom novih resursa, skalabilnost i mogućnost da platimo samo one resurse koje zaista trošimo.

2. MIKROSERVISNA ARHITEKTURA

Mikroservisni arhitektonski stil u razvoju softvera podrazumeva razvoj softvera kao skup malih servisa, pri čemu se svaki servis izvršava na svom procesu i komuniciraju koristeći protokole kao što su *HTTP/HTTPS, Web sockets* ili *Advanced Message Queuing Protocol* (skr. AMQP) [2]. Mikroservisi [3] se definišu kao mali autonomni servisi koji rade zajedno. Ovaj arhitektonski stil veoma brzo preovlađuje u industriji kao održiva alternativa monolitnim aplikacijama i servisno-orijentisanoj arhitekturi (eng. *service-oriented acrhitecture*, skr. SOA). Prednost ovakvih servisa je sto su usko povezani sa poslovnom funkcijom koju treba da izvršavaju [4].
Razlike između monolitne i mikroservisne arhitekture su prikazane na slici 1.



Slika 1. *Razlike monolitne i mikroservisne arhitekture* [11]

Monolitna aplikacija predstavlja funkcionalnosti specifične za domen i podeljene su uglavnom po nivoima kao što su *web*, poslovna logika, podaci (1). Skaliranje takve aplikacije radimo tako što je kloniramo na više servera / virtualnih mašina (2). Suprotno tome kod mikroservisne arhitekture svaka nezavisna poslovna logika predstavlja jedan servis (3). Skaliranje aplikacije prema mikroservis arhitektonskom stilu podrazumeva isporuku svakog servisa posebno kreiranjem instanci servisa na više servera / virtualnih mašina (4) [11].

3. AKTER PROGRAMSKI MODEL

Akter (eng. *Actor*) programski model predstavlja matematički model konkurentnog izračunavanja koji predstavlja aktere kao osnovne jedinice konkurentnog izračunavanja. Pojam je prvi put definisan 1973 u radu [5] gde je akter predstavljen kao aktivan agent, koji se ponaša u skladu sa dobijenim naznakama. Kao odgovor na poruku koju dobije, akter može da: donese logičku odluku, kreira nove aktere, pošalje poruku nekom drugom akteru ili da odluči šta će se desiti sa narednom porukom koju dobije. Akteri mogu da menjaju svoje privatno stanje, dok stanje drugog aktera mogu da menjaju samo razmenom poruka. Aplikacije srednjeg nivoa, odnosno nivoa poslovne logike, su najčešće objektno-orijentisane, jer uključuju interakciju sa velikim brojem objekata iz realnog svega. Primer su višekorisničke igre, društvene mreže, telemetrija. Kako su objektni iz realnog sveta nezavisni, tako i softverski objekti koji treba da ih modeluju ne treba da dele stanje i potrebno je da komuniciraju asinhronom razmenom poruka [6]. Takvi objekti se nazivaju akteri.

4. PRIMENA AKTER PROGRAMSKOG MODELA NA PROBLEM SKLADIŠTENJA PODATAKA

U [6] i [7] se predstavlja ideja da sloj za upravljanje podacima treba da bude apstrakcija akter modela kako bi se postigla visoka mogućnost skaliranja, konkurentnost i modularnost.

Enkapsulacijom stanja, podržavanjem *single-threaded* semantike za manipulaciju stanjem i asinhronom razmenom poruka, akter model predstavlja generalan, modularan i konkurentan model za izračunavanje. Pored toga izdržljivost i globalnu konzistentnost između aktera i upravljanje otkazima sistema nije obezbeđena od strane samog modela, već je ostavljena korisniku da je sam obezbedi. Pored ovih nedostataka, mnogi radni okviri kao što su Akka, Erlang i Orleans su često korišćeni za kreiranje sistema za skladištenje podataka u memoriji, dizajnirani tako da se skaliraju na milione aktera distribuiranih na stotine servisa korišćenjem *Cloud* infrastrukture [7]. U [6] i [7] su predstavljena načela koja opisuju upravljanje podacima primenom akter programskog modela. Pomenuta su samo neka od predstavljenih načela.

Za modularnost akter orijentisani sistem za skladištenje podataka u memoriji treba da podržava apstrakciju konkurentnog i distribuiranog logičkog aktera. Logički akter predstavlja entitet koji komunicira razmenom poruka. Logički akter se definiše kao entitet koji obrađuje zahtev prosleđen od strane klijenta ili drugog aktera. Akter model upravljanje životnim ciklusom aktera pruža samoj aplikaciji. Kod sistema kao sto je sistem za skladištenje podataka u memoriji, potrebno je omogućiti različite mehanizme za upravljanje životnim ciklusom aktera, kao što su dinamičko i statičko kreiranje aktera [7].

Kako bi se obezbedio brži odziv na zahteve, asinhrona razmena poruka postaje obavezna. *Single-threaded* semantika akter modela omogućava prilagođavanje klasičnih mehanizama baza podataka kao što su npr. transakcije [7]. Transakcija obuhvata više aktera koji se nalaze uglavnom na različitim serverima i nezavisno upisuju svoje stanje u skladište podataka. Zbog toga atomičnost i izdržljivost zahtevaju korišćenje transakcionih protokola [6].

Pozivanje metode aktera zahteva da je njegov identitet poznat. Poželjno je da je identitet aktera definisan njegovom lokacijom. Identitet može biti trajan, kao npr. primarni ključ u tabeli baze podataka ili može biti prolazan, odnosno da se prilikom aktivacije aktera kreira novi [6].

Pored toga što se akteru pristupa preko njegovog identiteta, korisno je da postoji mogućnost pristupa i preko drugih atributa [6], [7], [8]. U slučaju skladištenja istorijskih podataka u akter orijentisanu bazu u memoriji, dodatno ubrzanje prilikom pretrage može se postići indeksiranjem aktera prema datumu.

Ako bi se akteri tretirali kao objekti, objektno-orijentisan upitni jezik može biti primenjen na akter-orijentisane sisteme. Ako bi se akter-orijentisana baza predstavila kao graf, gde su čvorovi akteri, a grane reference između aktera, tada upiti mogu biti usmereni (eng. path-oriented), kao npr. SPARQL regularni izrazi [9].

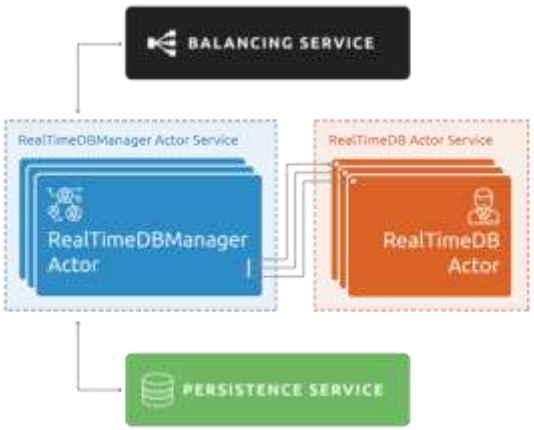
5. PREDLOG IMPLEMENTACIJE REŠENJA

5.1. Arhitektura predloženog rešenja

Osnovna ideja predloženog rešenja jeste podela podataka na određeni broj aktera. U opštem slučaju postoje menadžer (koordinator) akteri koji mogu predstavljati table u bazi podataka ili particionisanu tabelu u bazi podataka. Pored menadžer aktera, postoje i akteri koji su odgovorni za skladištenje podataka odgovarajućeg menadžer aktera.

Kao rešenje problema skladištenja istorijskih podataka, kreiran je distribuirani servis za pametno skladištenje podataka. Servis je implementiran uz oslonac na akter programski model upotrebom *Microsoft Azure Service Fabric* platforme. Sastoji se od *RealTimeDBManager* akter servisa, *RealTimeDB* akter servisa, *Balancing* servisa i *Persistance* servisa, prikazani na slici 2.

RealTimeDBManager akter servis dobija podatke odgovarajućeg tipa podataka i služi kao koordinator za skladištenje istorijskih podataka tog tipa. Akteri se identifikuju preko odgovarajućeg identifikatora. Svaki od *RealTimeDBManager* aktera ima reference na određeni broj *RealTimeDB* aktera koji skladište podatke. Pored skladištenja podataka u *RealTimeDB* akterima, *RealTimeDBManager* akter je odgovoran i za dobavljanje podataka od *RealTimeDB* aktera u određenom vremenskom periodu upotrebom asinhrone razmene poruka.



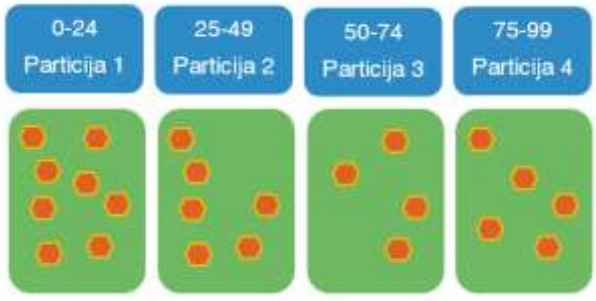
Slika 2. Arhitektura predloženog rešenja

5.2. Particionisanje podataka

Particionisanje predstavlja jedan od osnovih principa prilikom kreiranja skalabilnih servisa. Particionisanje se može posmatrati kao podela stanja i proračuna na manje jedinice kako bi poboljšali skalabilnost i performanse. Jedan od oblika particionisanja jeste particionisanje podataka (eng. *sharding*).
Kako je svaki akter servis, ustvari *stateful* servis, svaka particija akter sevisa će sadržati određeni skup aktera. Particija *stateful* servisa se može posmatrati kao skalabilna jedinica koja je dostupna preko svojih replika koje su distribuirane i balansiranje na nodovima (virtualnim mašinama). *Stateful* servisi mogu biti kreirani sa različitim šemama za particionisanje podataka.

5.3. Primena algoritma za particionisanje brojeva na problem particionisanja aktera

Akter servisi koriste isključivo *Int64* particionisanje sa upotrebom celog *Int64* opsega za vrednosti ključeva prilikom particionisanja.
Kada se kreira akter, potrebno mu je dodeliti odgovarajući jedinstveni identifikator reprezentovan *ActorId* klasom. Moguće je koristiti korisnički definisane identifikatore, kao što su GUID, UUID, stringovi i *Int64* vrednosti. Kada se koristi GUID, UUID ili string, identifikator se konvertuje u *Int64* opseg. Kada se koristi eksplicitno *Int64* vrednost kao identifikator aktera, tada će se taj akter automatski mapirati na odgovarajuću particiju bez prethodnog konvertovanja. Na ovaj način se može uticati na kojoj particiji će se akter kreirati [11].
Za potrebe ravnomerne raspodele aktera na particije korišćen je algoritam za rešavanje problema particionisanja brojeva (eng. *Number partitioning problem, skr*. NPP). Problem particionisanja brojeva jeste da dati skup brojeva podeli na međusobno isključive podskupove tako da je suma brojeva u svakom podskupu jednaka ili približno jednaka [10]. Predlog imlementiranog rešenja koristi pohlepnu heurstiku za proračun smeštanja aktera na particije.
Rad algoritma će biti objašnjen na primeru koji ilustruje slika 3. Kako bi se jednostavnije objasnio algoritam, koristiće se *Int64* šema za particionisanje sa minimalnim ključem 0, a maksimalnim 99. Postoje 4 particije, vrednosti ključeva iz opsega su ravnomerno raspodeljene na particije 0-24, 25-49, 50-74 i 75-99.

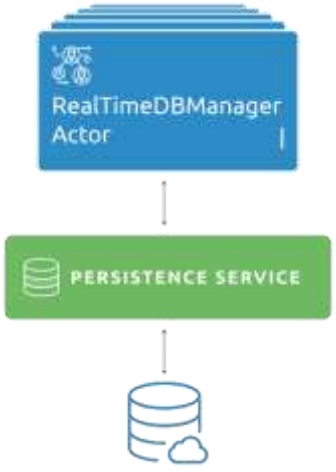


Slika 3. Particionisanje aktera

Kada *RealTimeDBManager* akter treba da kreira novi *RealTimeDB* akter, šalje poruku *Balancing* servisu, koji odredi koja je particija najmanja i prosledi *RealTimeDBManager* akteru sledeću vrednost iz opsega ključeva za tu particiju. U situaciji prikazanoj na prethodnoj slici, particija 3 ima najmanji broj kreiranih aktera (id aktera na ovoj particiji su 50, 51, 52, 53), tako da će *Balancing* servis vratiti prvi naredni ključ iz opsega ključeva koji odgovara toj particiji, a to je ključ 54.

5.4. Dostupnost i pouzdanost podataka

Kritičan deo u obezbeđivanju visoke dostupnosti jeste obezbediti otpornost servisa na otkaze. Kako bi se zaštitili od mogućeg gubitka podataka potrebno je periodično kopiju podataka skladištenih u memoriji, skladištiti na nekom trajnom skladištu podataka, na primer relacionoj bazi podataka koja podatke skladišti na disk. Do gubitka podataka u memoriji može doći iz više razloga.
Otkaz pojedinih nodova na *Service Fabric* klasteru može dovesti do gubitka servisa na tom nodu, kao i gubika stanja tih servisa. U takvom slučaju poželjno je posedovati više replika servisa. Implementirano rešenje podrazumeva upotrebu *Service Fabric Reliable Services* replikacionog mehanizma [11]. Svaki od kreiranih aktera predstavlja primarnu repliku tog aktera. Pored primarne replike postoje i dve sekundarke replike od koje jedna postaje primarna u slučaju otkaza primarne replike.
Samo u slučaju inicijalizacije servisa za skladištenje podataka u memoriji potrebno je vršiti čitanje podataka iz relacione baze podataka, u suprotnom RAM memorija treba da bude primarno skladište podataka u koju se podaci unose i iz koje se čitaju.
Slika 4 prikazuje perzistenciju podataka aktera na *Azure SQL* bazu podataka. Kada dobije podatak, *RealTimeDBManager* akter šalje taj podataka na *Persistance* servis koji ima ulogu skladištenja podataka u relacionoj bazi. Pored toga što vrši skladištenje podataka u bazu podataka, *Persistance* servis vrši i inicijalizaciju podataka u memoriji prilikom startovanja servisa. Učitava podatke iz relacione baze i šalje ih na *RealTimeDBManager* akter servis, koji potom vrši raspodelu podataka na *RealTimeDB* aktere. Da bi *Persistance* servis mogao da čita podatke iz baze, mora da zna koje tipove podataka treba da učita iz baze podataka. Skup tipova podataka se inicijalno čita iz baze podataka, a ažurira se prilikom čuvanja novog tipa podataka u bazu.



Slika 4. Perzistencija podataka

6. PERFORMANSE REŠENJA

Vršena su merenja performansi sistema. Za testni skup podataka korišćena je relaciona baza koja sadrži oko 9000 istorijskih merenja snaga po generatoru. Merenja performansi sistema su izvršena na lokalnom klasteru, računaru konfiguracije: CPU i5 2.20GHz, RAM 12GB. Klaster sadrži 5 nodova. *RealTimeDB* akter servis je particionisan na 3 particije, dok je *RealTimeDBManager* akter servis particionisan na 2 particije.
Performanse su merene za dva slučaja u kojima je broj kreiranih aktera različit. Tabela 1 i tabela 2 sadrže podatke o broju entiteta koji su dobavljeni prilikom upita, vreme trajanja izvršavanja upita na implementiranom rešenju (M) i vreme izvršavanja upita na relacionoj bazi podataka (DB). Tabela 1. predstavlja rezultete vremena dobavljanja svih podataka po generatoru u situaciji kada je maksimalan broj entiteta koje akter može da skladišti 100.

ID	1	2	3	4	5	6	Avg
Ent	9075	9300	9259	8509	8527	9300	8995
M [ms]	398	458	434	405	436	395	421
DB [ms]	1791	664	1196	1263	1325	2311	1425

Tabela 1. Vremena dobavljanja svih merenja po generatoru – 100 entiteta po akteru

Tabela 2. predstavlja rezultete vremena dobavljanja svih podataka po generatoru u situaciji kada je maksimalan broj entiteta koje akter može da skladišti 50. U ovom slučaju je ukupan broj aktera u sistemu duplo veći nego u prethodnom slučaju. Što znači da prilikom pretrage merenja po određenom kriterijumu (vremenskom opsegu) sistem treba da pretraži duplo veći broj aktera i zbog toga je vreme potrebno za izvršavanje upita veće u odnosu na prethodni slučaj.

Tabela 2. *Vremena dobavljanja svih merenja po generatoru – 50 entiteta po akteru*

ID	1	2	3	4	5	6	Avg
Ent	9075	9300	9259	8509	8527	9300	8979
M [ms]	851	816	986	882	976	865	905
DB [ms]	2194	2152	2032	1985	1926	2216	2062.2

Na lokalnom klasteru, rad servisa se izvršava konkurentno. Na *Cloud* klasteru izvršavanje servisa na različitim nodovima je paralelno i zbog toga bi veliki broj particija na kojima se upiti izvršavaju paralelno omogućio brže performanse. Performanse zavise od fizičkih karakteristika samih nodova, kao i od broja nodova na klasteru.

7. ZAKLJUČAK

U radu je predstavljeno rešenje skladištenja podataka u memoriji upotrebom akter programskog modela i mikroservisnog arhitektonskog šablona. Analizirani su radovi koji se bave pomenutom tematikom i neki predlozi rešenja predstavljeni su u ovom radu. Na osnovu nekih od tih predloga implementirano je osnovno softverko rešenje koje omogućava particionisanje podataka na veliki broj aktera koji u svojoj memoriji skladište podatke. Problem ravnomernog iskorišćenja resursa rešen je upotrebom algoritma za particionisanje brojeva.
Kako bi se obezbedila pouzdanost podataka, realizovano je repliciranje podataka između aktera, gde postoji jedna primarna instanca svakog aktera i dve sekundarne replike, i perzistencija podataka na *Azure SQL* bazu podataka.

8. LITERATURA

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," *Nist Spec. Publ.*, vol. 145, p. 7, 2011.
[2] C. de la Torre, B. Wagner, and M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*, V1.0. Redmond, Washington: Microsoft Corporation.
[3] S. Newman, *Building Microservices*, 1st ed. Sebastopol, CA: O'Reilly media, Inc., 2015.
[4] M. Fowler and J. Lewis, "Microservices." [Online].Dostupno:https://martinfowler.com/articles/microservices.html. [Pristupljeno: 12-sep-2018].
[5] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence, 1973, pp. 235–245.
[6] P. Bernstein, "Actor-Oriented Database Systems," in *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*, 2018, pp. 13–14.
[7] V. Shah and M. A. V. Salles, "Actor Database Systems: A Manifesto," no. 1, 2017.
[8] P. A. Bernstein, T. Kiefer, and D. Maier, "Indexing in an Actor-Oriented Database," *Conf. Innov. Database Res.*, 2017.
[9] P. Hitzler, M. Krötzsch., and S. Rudolph, *Foundations of Semantic Web Technologies*. 6000 Broken Sound Parkway NW, Suite 300: Chapman & Hall/CRC textbooks in computing, 2010.
[10] R. E. Korf, "Multiway Number Partitioning," *Proc. Twenty-First Int. Jt. Conf. Artif. Intell.*, pp. 538–543, 2009.
[11] "Introduction to microservices on Azure | Microsoft Docs." [Online]. Dostupno: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices. [Pristupljeno: 23-sep-2018].

Kratka biografija:

Marko Jelača rođen je 12.09.1994. godine u Sremskoj Mitrovici. Završio je Mitrovačku gimnaziju u Sremskoj Mitrovici 2013. godine. Fakultet tehničih nauka u Novom Sadu je upisao 2013. godine, a diplomirao na smeru Elektroenergetski softverski inženjering 2017. Iste godine upisuje master studije na smeru Primenjeno softversko inženjerstvo.