Univerzitet u Beogradu

Matematički fakultet

Aleksandar Minić

Obrasci za projektovanje za računarstvo u oblaku

Master rad

Beograd

2017.

Univerzitet u Beogradu - Matematički fakultet

Master rad

Autor:	Aleksandar Minić
Naslov:	Obrasci za projektovanje za računarstvo u oblaku
Mentor:	prof. dr Saša Malkov
Članovi komisije:	prof. dr Vladimir Filipović
	prof. dr Filip Marić
Datum:	**--TODO datum odbarne**

Sadržaj:

1. Uvod

1.1 Računarstvo u oblaku

Računarstvo se, od svog nastanka, razvija velikom brzinom. Gordon Mur, jedan od osnivača *Intel*-a, 1965. godine je definisao zakon po kome je utvrdjen eksponencijalni rast računarstva, koji važi i dan danas. Ovaj zakon, poznatiji i kao Murov zakon[1], tvrdi da broj tranzistora koji se može smestiti na integrisanu matičnu ploču raste eksponencionalno, odnosno udvostručuje se svakih 18-24 meseca. Ovo predviđanje pokazalo se ne samo tačnim u periodu definisanja, već važi i dan danas, više od 50 godina kasnije.

Na Murov zakon, kasnije su se nadovezali i drugi zakoni, koji su primetili sličan trend i u drugim oblastima. Tako je Mark Krajder primetio i definisao sličnu pravilnost vezanu za kapacitet diskova za skladištenje podataka. U Krajderovom zakonu[2] se definiše da se kapacitet tvrdih diskova duplira na svakih godinu dana. U *Bell* labaratorijama su definisali zakon koji se tiče kapaciteta mreže, po kome se na svakih devet meseci cena slanja jednog bita preko optičke mreže prepolovi. Ovaj zakon poznat je i kao Baterov zakon[3].

Ovakav trend razvoja doveo je do ogromnog rasta i zastupljenosti samog računarstva. Računarstvo postaje sve prisutnije u svakodnevnom životu, tako da se sada koristi u raznim oblastima, od astronomije, industrije, farmacije, zdravstva, do svakodnevnih aktivnosti poput zabave, transporta i slično. Pravilnost koju je uočio Murov zakon pomogao je da sve ove oblasti koriste računare u svakodnevnom radu, kao i da definišu svoje probleme i zadatke koji ranije nisu bili mogući ili optimalni, koristeći računare kao osnovni alat. Rastom moći računara njihova cena je padala i samim tim postala dostupnija za mnoge kompanije, istrazivačke centre, pa i same pojedince. Međutim, kako je Murov zakon obezbedio jeftinije računare, kao posledicu je imao da i sami računari jako brzo postanu zastareli i da je neophodno svakih par godina ulagati u nove računare kako bi se pratio razvoj računarstva i potrebe potrošača i korisnika. Samo uvođenje novog hardvera prati velike troškove, migracije, redizajniranje sistema i slično. Hardver koji je nabavljen po ogromnim cenama pre desetak godina, danas je praktično neupotrebljiv.

Veliki računarski centar se može dizajnirati na dva načina:

- Optimizacija troškova: Centar je dovoljno dobar za veći deo zadatka koji rešava i dostupan je veliki deo vremena. Takav sistem će raditi u velikom procentu slučajeva koji se stavljaju pred njega, i korisnicima će servisi biti dostupni. Međutim, kada broj korisnika postane prevelik, odnosno u vršnom opterećenju sistema, takav sistem neće

moći da odgovori svim zahtevima. Prosečna iskorišćenost ovakvih sistema je obično od 5 do 20 procenata[4].

- Optimizacija dostupnost: Dizajniranje sistema na način da on uvek ima dovoljno resursa da zadovolji sve zahteve, čak i pod najvećim opterećenjem. To je moguće postići postavljanjem sistema prema maksimalnom broju korisnika, odnosno projektovanje sistema da uvek ima dovoljno resursa da zadovolji sve zahteve, iako se to dešava jako retko. Iskorišćenost ovakvih sistema je loša, i vrlo često može da ide i ispod jednog procenta.

Troškovi održavanja ovakvih sistema mogu biti veoma visoki, tako da pored cene samog računarskog centra veliki izdatak predstavljaju i naknade za struju, prostor, sigurnosne i prateće sisteme. Međutim, možda i glavna mana ovih sistema jeste neelastičnost. Odnosno, širenje kapaciteta ovakvog sistema zahteva velike resurse, a vrlo često je i neizvodljivo. Pored cene samog hardvera i potrebne podrške za rad istih, treba uzeti u obzir da je za širenje sistema potrebno skalirati i prateću infrastrukturu, sisteme, promenu procedura koje se sprovode i slično.

Kompanije poput *Amazon*-a, *Google*-a, *Microsoft*-a su uvidele da veliki broj njihovih sistema veći deo vremena nema dovoljno posla, odnosno da nisu dovoljno iskorišćeni. S obzirom da se njihovi sistemi projektuju da uvek mogu da izdrže najveće opterećenje, takozvano vršno opterećenje, preostalo vreme ti sistemi stoje prazni, neiskorišćeni. Ova činjenica ne samo da je inspirasala pokretanje računarstva u oblaku, već je dala i pun zamajac razvoju istog. Poslužioci su mogli da daju na korišćenje računare koje ne koriste sve vreme. Oni su ponudili svoju ekspertizu u izradi velikih, skalabilnih sistema i pružili korisnicima niz funkcionalnosti koje su upakovali u svoj jedinstveni proizvod, nazvan računarstvom u oblaku.

Računarstvo u oblaku se obično deli na tri vrste. Infrastruktura kao servis, odnosno *IaaS (Infrastructure-as-a-Service)*, je oblak koji pruža usluge iznajmljivanja hardvera sa pratećim softverom na određeno vreme. Poslužioc ovakvog oblaka pruža jako malo usluga osim samog održavanja centra podataka operativnim. Korisnici su zaduženi za održavanje softvera na isti način kao što bi to radili u svojim centrima podataka. Ukoliko poslužioc pored hardvera ponudi i deo softvera koji bi pomogao integraciji u druge sisteme, onda govorimo o platformi kao servisu, ili *PaaS (Platform-as-a-Service)*. *PaaS* nudi skup servisa programerima koji olakšavaju razvoj aplikacija preko veba, bez kompleksnosti kupovine i održavanja infrastrukturnog sloja. Softver kao servis, ili *SaaS (Software-as-a-Service)* je ideja da poslužioc ponudi skup softvera koji se nalaze na serverima poslužioca i koji se plaćaju po iskorišćenosti. Korisnik nema potrebe za razvojem ili održavanjem softvera, on može da ga koristi preko veba i jedino što je neophodno da zna jeste da korsiti sam softver.

Danas računarstvo u oblaku koriste mnogi. Od IT profesionalaca i poslovnih menadžera do naučnika i istraživača. Svaka od ovih grupa definiše računarstvo u oblaku na osnovu onoga što im ono pruža. Iako ne postoji sveprihvaćena definicija računarstva u

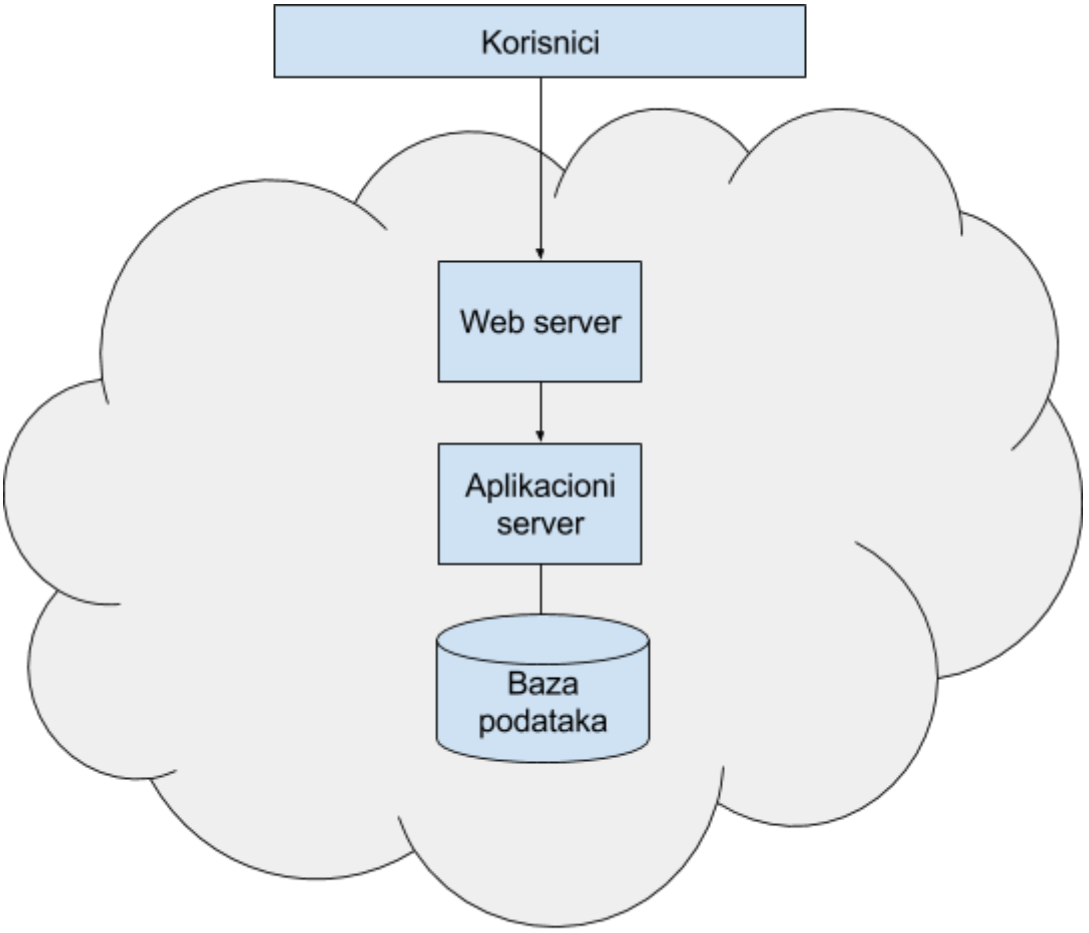
oblaku, ono ipak nudi obećavajuću paradigmu koja može omogućiti poslovima i sistemima da odgovore promenama na tržištu na agilan i troškovno efikasan način. U poslednje vreme javlja se konzistentna definicija oblaka:

> „Prenošenje posla vezanog za IT aktivnosti na jednu ili više trećih strana koje imaju velike zalihe resursa da zadovolje potrebe organizacije na lak i efikasan način. Ove potrebe mogu uključivati hardverske komponente, mreže, skladištenje podataka i softverske sisteme i aplikacije. Dodatno, oni mogu da uključuju i infrastrukturne elemente kao što su fizički prostor, opremu za hlađenje, struju, protivpožarne sisteme i ljudske resurse za održavanje tih sistema”[5].

Osnovu računarstva u oblaku predstavlja iznajmljivanje samih servera i servisa, koje drugi mogu da zakupe na određeno vreme, i tako plate za korišćenje istih samo onoliko koliko ih koriste. Naravno, sve ovo je postalo isplativo za obe strane pre svega zahvaljujući virtualizaciji, jer je time bilo moguće smestiti više virtuelnih servera na jednu fizičku mašinu, i tako maksimalno iskoristiti mašine koje poslužioci oblaka imaju. Od osnovnog sistema iznajmljivanja hardvera, otišlo se daleko. Iako i danas to čini osnovu svih oblaka, sistemi su otišli mnogo dalje, i sada nude i prostore za skladištenje podataka, predefinisane sisteme za lakše skaliranje, baze podataka koje je lako podesiti i izmeniti da podnesu veliko opterećenje, i mnogo drugih stvari.

1.2 Osnovna arhitektura u oblaku

Najprimamljiviji način prelaska u oblak je da se celokupan računarski sistem prenese iz lokalnog centra u oblak. Na taj način se ostvaruje ušteda na hardveru bez većih zahvata na prilagođavanju softvera.



Slika 1: Osnovna arhitektura u oblaku

Arhitektura sistema ostaje identična onoj koja je bila, i na slici 1 je prikazan takav sistem koji se ne razlikuje od prethodne postavke osim što je sistem smešten u oblaku. Na ovaj način je jednostavno pokrenuti svoj sistem, bez kupovine hardvera. Softer koji je korišćen ostaje isti, a ostaje i mogućnost da se svaki server pojača, odnosno zameni jačim hardverom. Naime, ukoliko bi se vremenom ustanovilo da je neki server prezauzet i da je potrebna jača mašina koja bi mogla da odgovi većem broju zahteva, poslužioci oblaka su ponudili način za dinamičko povećanje instanci[6, 7, 8].

Sam proces pojačavanja servera je pravolinijski kod svih većih poslužioca poput Google-a, Amazon-a i Microsoft-a. Povećanje instance se radi tako što se server zaustavi i onda odabere novi tim servera koji odgovara potrebama sistema. Svi podaci koji su bili na serveru ostaju i na novom, jačem, bez potrebe za migraciom. Nakon pokretanja servera sistem nastavlja da radi samo sa većim mogućnostima. Za razliku od tradicionalnih sistema, gde bi ovakva migracija podrazumevala novi hardver, u oblaku se u svakom trenutku naplaćuje samo oni servisi koji se koriste, i na taj način troškovi su vrlo često dosta manji. Sistem je kratko nedostupan, ali zato se potencijalno dobija dosta na skalabilnosti.

1.2 Obrasci za projektovanje

Za definiciju obrazaca za projektovanje, većina se oslanja na definiciju koju je dao Kristofer Aleksander u svojoj knjizi „Jezik obrazaca: gradovi, zgrade, konstrukcije". On obrasce definiše na sledeći način:

> „Svaki obrazac opisuje problem koji se iznova i iznova ponavlja u našem okruženju, zatim opisuje osnovu rešenja za taj problem na takav način da se opisano rešenje može koristiti još milion puta, tako da se nikada ne ponavlja isti način izrade iz početka, više puta"[9].

Naravno, Kristofer je ovde mislio na građevinske obrasce, koji služe za projektovanje gradova i zgrada, ali ista definicija je primenljiva i na računarske sisteme.

Na Kristoferov pogled na obrasce se nadovezuje i Linda Rajzing, koja u knjizi „Priručnik o obrascima"[10] kaže da je Aleksanderova definicija zasnovana na spoznaji da je razlog postojanja ponavljajućih rešenja zapravo postojanje ponavljajućih problema. S toga, podjednak deo obrasca jesu i problemi koje obrazac rešava, kao i odgovarajuć raspored objekata koji čine rešenje.

Obrasci treba da pomognu korisnicima da ne prolaze iste, česte, probleme iznova i iznova. Ono što je jednom napisano i pokazalo se kao dobro rešenje u praksi treba koristiti ponovo za rešavanje sličnih problema. Uočavanje samog problema je prvi korak ka obrascu. Sam problem i jeste zapravo deo obrasca jer on opisuje šta je neophodno rešiti. Nakon što je uočen problem, onda je potrebno naći obrazac koji se bavi istim tim problemom. Samo rešenje problema u obrascu je opštih karakteristika, odnosno opisano je da na apstraktnom nivou objasni kako ga iskoristiti. Naravno deo svakog obrasca jeste i kritički pogled na problem i rešenje, kao i uputstvo kada obrazac možda ne treba koristiti, ili imati u vidu moguće loše strane tog rešenja.

Svaki obrazac ima ime koje ukratko, vrlo često na simboličan način, opisuje šta ili kako taj obrazac radi. Na osnovu njega lako ga je razlikovati, jednostavnije je pričati o njemu, kritikovati ga i razmenjivati mišljenja o istom. Poznati obrasci su na primer Graditelj, koji pravi različite objekte po zahtevu, Unikat, koji predstavlja način da uvek postoji samo jedna instanca objekta koju svi koriste. Svakako, na osnovu imena lako je pretpostaviti šta svaki od ovih obrazaca radi, ali uz ime svakog obrasca obavezno je opisati i kojim problemom se bavi. Ukoliko je logika pravljenja nekog objekta kompleksna, i potrebno je izdvojiti logiku pravljenja samog objekta, jasno je da se problem koji postoji rešava obrascem Graditelj. Tako obrazac Unikat rešava problem konkurentnog pristupa objektu, koji zapravo uvek može imati samo jedno stanje. Nakon definisanja problema svaki obrazac opisuje kako se taj problem

rešava, primenom različitih tehnologija, mehanizama i drugih obrazaca. I na kraju, svaki obrazac nosi neko kritičko mišljenje, kada i kako ga treba ili ne treba koristiti.

2. Obrasci za projektovanje u oblaku

2.1 Specifičnost problema projektovanja u oblaku

Svi obrasci koji su obrađivani i korišćeni u tradicionalnim sistemima nastavljaju da budu primenljivi, u manjoj ili većoj meri, i u sistemima koji se nalaze u oblaku. Svi oni donose značajna poboljšanja, ali kako je u oblaku okruženje drugačije javljaju se novi problemi koje je potrebno rešiti. S obzirom da se veliki deo tih problema ponavlja, oni su zajedno sa svojim rešenjima formirali nove obrasce primenljve u oblaku. Obrasci opisani u ovom radu nisu obrasci koji se isključivo koriste u oblaku, već oni obrasci koji rad i razvoj sistema u oblaku čine lakšim, a sistem boljim.

Oblak nudi mnogo novih mogućnosti. Moguće je dinamički skalirati, kako u pogledu veličine servera tako i u pogledu broja servera. Sa porastom broja servisa raste i broj obrade tih zahteva, pa se menja i način na koji se oni obrađuju. S povećanjem mogućnosti raste i povećanje kompleksnosti sistema, pa se stvaraju i veće potrebe za nadgledanjem rada sistema i njegovim oporavkom. Svi ovi problemi su se izdvojili u nove obrasce, koji su od velikog značaja u većini sistema u oblaku. Njihova implementacija povećava kapacitet sistema, vreme dostupnosti i stabilnost. Kako računarstvo u oblaku biva sve zastupljenije, pojavljuju se i novi načini dizajniranja sistema kao i novi pristupi u rešavanju tradicionalnih problema.

Svi obrazci koji su opisani u ovom radu su plod istraživanja rada računarstva u oblaku, preporuka poslužilaca oblaka[11], radova drugih autora[12], kao i sopstvenog iskustva u radu na sistemima u oblaku. Problemi koji su opisani su neki od najčešće opisivanih i otkrivenih, i zajedno daju dobar pregled problema i rešenja sa kojima se sistem u oblaku suočava.

Svaki od obrazaca bi trebalo da sadrži četri osnovna dela[13] ukoliko je to moguće:

- **Ime:** Naziv koji služi da opišemo problem, njegovo rešenje i posledice i to na sažet način u par reči.

- **Opis problema:** Objašnjava problem na koji nailazimo kada opisujemo neki obrazac. Objašnjava problem, pod kojim uslovima se javlja.
- **Rešenje:** Opisuje delove koji čine opis, njihove odnose, odgovornosti i međusobnu saradnju. Rešenje ne daje kompletnu implementaciju problema jer obrazac čini zapravo šablon koji opisuje na koji način problem možemo rešiti i kako možemo primeniti obrazac u različitim situacijama.
- **Posledice:** One su rezultat ustupaka koji se moraju napraviti da bi se obrazac primenio. Sagledavanje posledica pomaže da bolje razumemo obrazac i da lakše odlučimo da li je svrsishodno implementirati isti. Kroz kritički stav ka obrascu lakše je razumeti i kako on radi, gde su njegovi nedostaci i da li smemo da dopustimo da iste nedostatke uvedemo u svoj sistem.

U radu će svi obrasci biti opisani sa nezavisnim pogledom na svaki od njih ali i kroz mogućnosti kombinovanja sa drugim obrascima kako bi se dobili bolji rezlutati.

2.2 Katalog obrazaca

2.2.1 Obrazac „Čuvar kapije”

Problem

Sistemi se sastoje od skupa servisa koji nude svoje funkcionalnosti korisnicima. Svaki od servisa često ima deo koda koji prima zahteve od korisnika, vrši proveru ispravnosti i da li je korisnik autentikovan. Ovaj deo koda proverava da li je korisnik zlonameran, da li ima pravo pristupa tim informacijama, koji je skup privilegija i slično. U velikom broj takvih sistema oni i odlučuju da li će uopšte i opslužiti zahtev. Pored toga, servisi se veoma često nalaze na serverima koji na sebi imaju neke poverljive podatke, poput sigurnosnih ključeva, sertifikata, lozinki za pristup bazi i slično. Ukoliko se takav servis kompromituje, napadač može da pristupi ovim podacima i na taj način ugrozi ceo sistem.

Rešenje

Jedan od načina za rešavanje problema jeste korišćenje servisa koji će obraditi svaki zahtev koji dolazi od strane korisnika, i onda odlučiti da li je zahtev potrebno proslediti sistemu ili ne.

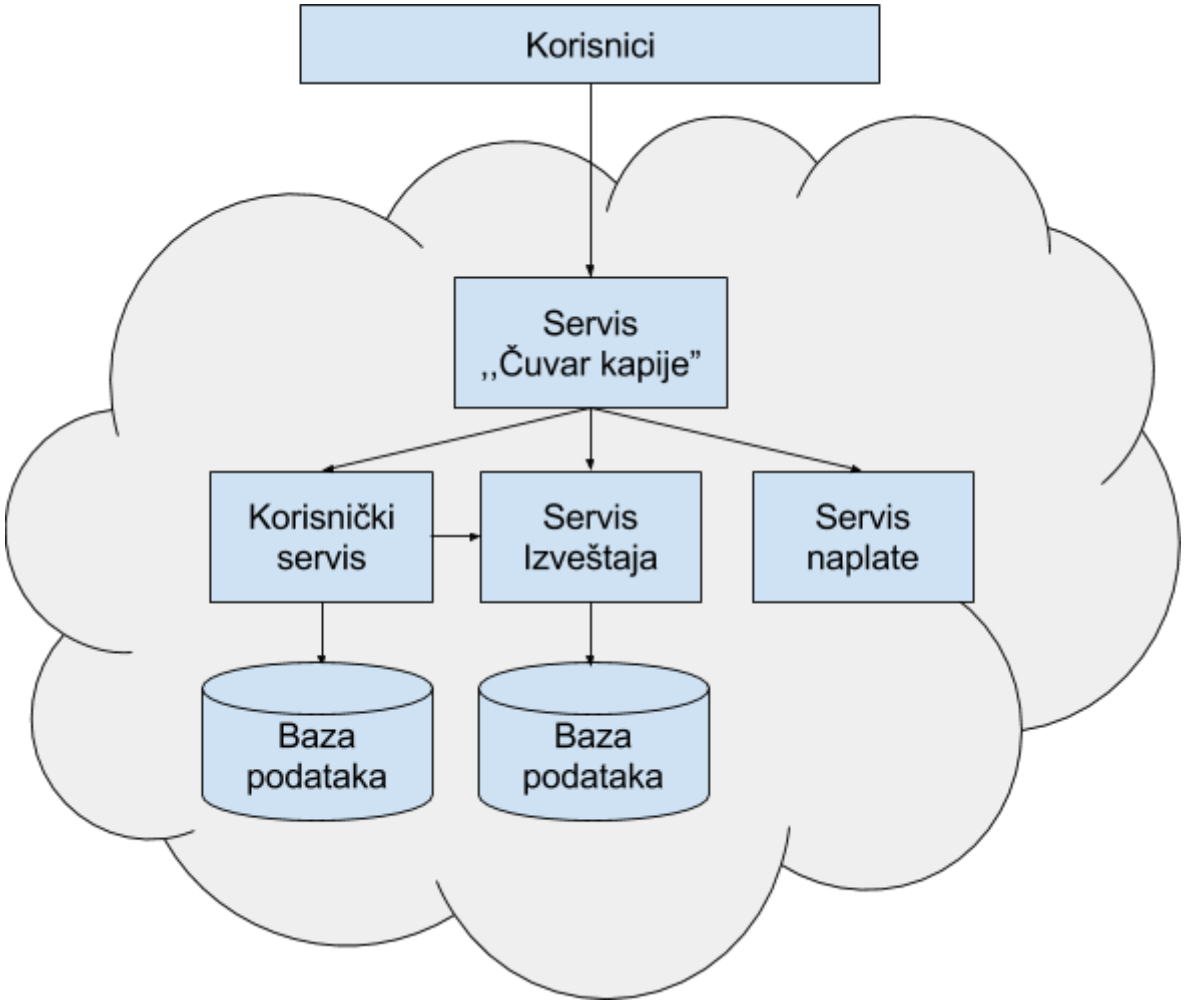
Jedini servis koji prima zahteve od korisnika, obrađuje ih i odlučuje da li ih želi proslediti ostalim servisima naziva se simbolično Čuvar kapije. Kod ovakve arhitekture Čuvar kapije analizira svaki zahtev. Ukoliko je potrebno on svaki od njih pročisti u smislu sklanjanja suvišnih informacija, zatim proveri skup dozvola koje korisnik ima i odlučuje da li prosleđuje zahtev sistemu na obradu ili ga odbacuje. Ovakvi servisi kada donose odluku o dozvolama za pristup uglavnom imaju svoju bazu koja sadrži sve podatke o korisnicima i privilegijama, mada mogu koristiti i eksterne servise koji se time bave. Ovaj servis se koristi isključivo za proveru ispravnosti zahteva, i nikakva obrada se ne radi u njemu, zbog performansi i kompleksnosti servisa.

Na slici 1 vidimo primer takvog sistema. Svi zahtevi od strane korisnika idu isključivo ka Čuvaru kapije. Ukoliko zahtevi zadovolje unapred definisane kriterijume, prosleđuju se onom servisu kojeg se tiču, odnosno koji je zadužen za njihovu obradu. Nijedan zahtev ne može i ne treba da pristupa servisima direktno. Svaki zahtev je neophodno da prođe preko Čuvara, koji je obavezan posrednik u komunikaciji.

Posledice

Ovaj obrazac pruža centalizovano mesto ispitivanja ispravnosti. On proverava sve zahteve i odbacuje one koji ne zadovoljavaju sve neophodne kriterijume. Na taj način imamo jedno mesto na kome je, eventualno, neophodno promeniti polise sigurnosti, i tako se mogućnost greške drastično smanjuje. Takođe, Čuvar kapije ima ograničen skup privilegija, i ne može da pristupi ostalim delovima sistema, osim preko već unapred definisanih kanala komunikacije.

Kako ovaj servis nema pristup ostalim bazama, sigurnosnim ključevima i slično, ukoliko dođe do eventulnog proboja u sistem, sistem nije kompromitovan. Svi podaci do kojih je napadač mogao da dođe ne mogu se iskoristi za dalji napad na ostale delove sistema.



Slika 1: Obrazac Čuvar kapije

Za Čuvara kapije je neophodno da se izvšava sa ograničenim skupom privilegija i neophodno ga je izolovati na posebnu virtuelnu ili fizičku mašinu. Naravno, ova mera sigurnosti zahteva nove resurse što dovodi do dodatnih troškova, kao i do povećanja kompleksnosti sistema.

Kako svi zahtevi prolaze kroz Čuvara kapije, to uvodi dodatan korak prilikom obrade svakog zahteva. Ovo može usporiti sistem i zato je veoma bitno da Čuvar bude napisan tako da ima što je moguće manje logike u sebi. Sva poslovna logika bi morala da stoji u posebnim servisima koji se bave tim domenom, a Čuvar da bude što jednostavniji. Takođe, s obzirom na centalizovanje sistema uvođenjem ovog obrasca, on može da bude takozvana jedinstvena tačka pucanja. Kada je on nedostupan, nedostupan je i ceo sistem. Takođe, ako svi zahtevi dolaze do njega, on mora biti u stanju i da izdrži veliki broj zahteva. Ovo se postiže nekim od obrazaca koji će biti obrađeni kasnije.

2.2.2 Obrazac ograničavanja broja zahteva

Problem

Kada postoji veliki broj zahteva, nije uvek moguće sve ih opslužiti. Ukoliko korisnik ne može da prosledi nijedan zahtev ka serveru, iako deluje da sistem radi, stvara se loše korisničko skustvo. Zahtevi, koje korisnik šalje, ostaju bez ikakvog odgovora od strane sistema. To se može desiti kada jedan od korisnika pošalje toliko zahteva da iskoristi sve raspoložive kapacitete sistema i niko drugi ne može da mu pristupi. Ceo sistem radi punim kapacitetom i opslužuje veliki broj zahteva ali, ti zahtevi mogu biti upućeni od malog dela korisnika, dok ostali korisnici nemaju najbolju sliku o tome šta se dešava sa sistemom i zašto ne dobijaju odgovore na zahteve.

Rešenje

Rešenje ovog problema je da se svakom korisniku unutar sistema definiše broj zahteva koje može da uputi u jednom trenutku ili periodu vremena, tako da sistem uvek ima dovoljno resursa da zadovolji i deo zahteva ostalih korisnika.

Ukoliko sistem može da podnese X paralenih zahteva, a u trenutku najvećeg korišćenja sistema procenjuje se da će imati po M zahteva od Y korisnika, takvih da je: $X < M*Y$, tada sistem treba konstrusati da nijedan korisnik ne može da pošalje više od M zahteva u definisanom periodu. Odnosno da sistem nakon primljenih M zahteva od jednog korisnika, ne prima nove zahteve, dok se neki od prethodnih ne završi ili prodje predefinisani period. Svaki zahtev nakon M-tog treba odbiti i obavestiti korisnika da je dostigao limit.

Ovaj obrazac ne mora biti usmeren samo na spoljašnje korisnike, već se može koristiti i u komunikaciji sa drugim delovima sistema. Odnosno treba utvrditi koji su delovi sistema manje bitni za vitalne funkcionalnosti, i ograničiti ih. Tako se obezbeđuje da vitalni delovi sistema uvek imaju dovoljno resursa da odrade posao koji je neophodan.

Klasičan primer korišćenja ovog sistema jeste broj zahteva koji se mogu proslediti na javni API. Ilustraciju ovog primera možemo videti na Primeru 1.

```
public Odgovor ograničiAPIPozive (SesijaKorisnika sk,
                                  APIPodaci podaci){
      Odgovor odgovor = new Odgovor();
      //dohvati koliko aktivnih zahteva korisnik ima
      Integer zauzetost = dohvatiZauzetost (sk.dohvatiKorisnika());
      if (zauzetost != null && zauzetost > LIMIT){
        odgovor.setValidan = false;
        odgovor.setPoruka = "Dotignut je limit broja zahteva korisnika"
      } else {
        //povecaj brojac zahteva za korisnika
        dodajZauzetost(sk.dohvatiKorisnika());
        //izvrsi zahtev
        odgovor = izvrsiAPIPoziv(sk, podaci);
      }
      return odgovor;
      }
```

Primer 1: Ograničavanje API poziva

Posledice

Kada je implementiran obrazac ograničavanja, ceo sistem može da odgovori na zahteve nezavisno od toga koliko drugi korisnici koriste sistem. S obzirom da se i prilikom prekoračenja limita dobija odgovor servera sa jasnom porukom, svi korisnički alati koji koriste sistem mogu da obrađuju taj odgovor. U skladu sa tim odgovorom oni mogu prilagoditi svoje ponašanje, odnosno svaka aplikacija može pokrenuti različite komande. Prilagođavanje ponašanja može biti različito, od ponovnog pokušavanja, slanja obaveštenja kako bi se limiti povećali i slično. Svakako, korisnicima se daje mogućnost da se sami prilagode ukoliko dosegnu limit, a serverska aplikacija ima dovoljno resursa da opsluži veliki broj korisnika.

Ovaj obrazac uvodi još jednu dodatnu proveru pre nego što prosledi zahtev na izvršavanje. Kako se time uvodi dodatno trajanje obrade zahteva, ceo sistem se ponaša sporije. Takođe, uvođenje ovakvog mehanizma je dosta komplikovanjije u kasnijim fazama razvoja sistema, i potrebno ga je uvesti u inicijalnim planovima razvoja sistema.

Ovaj sistem, međutim, može smanjiti iskorišćenost. Naime, ukoliko je priroda korišćenja sistema takva da se pušta veliki broj zahteva, ali samo od jednog korisnika u isto

vreme, ograničavanje može biti kontraproduktivno. Iako sistem ima kapaciteta da obradi sve zahteve korisnika, veliki broj zahteva će čekati da se prethodni završe. Sistem će odbaciti veliki broj zahteva, iako ima dovoljno resursa da obradi mnogo veći broj zahteva. Zapravo ovaj obrazac ima svoju pravu svrhu pre svega u sistemima sa velikim brojem korisnika u paraleli. On tada garantuje da će svi oni biti opsluženi, i da nijedan korisnik neće biti u povlašćenom položaju u odnosu na ostale.

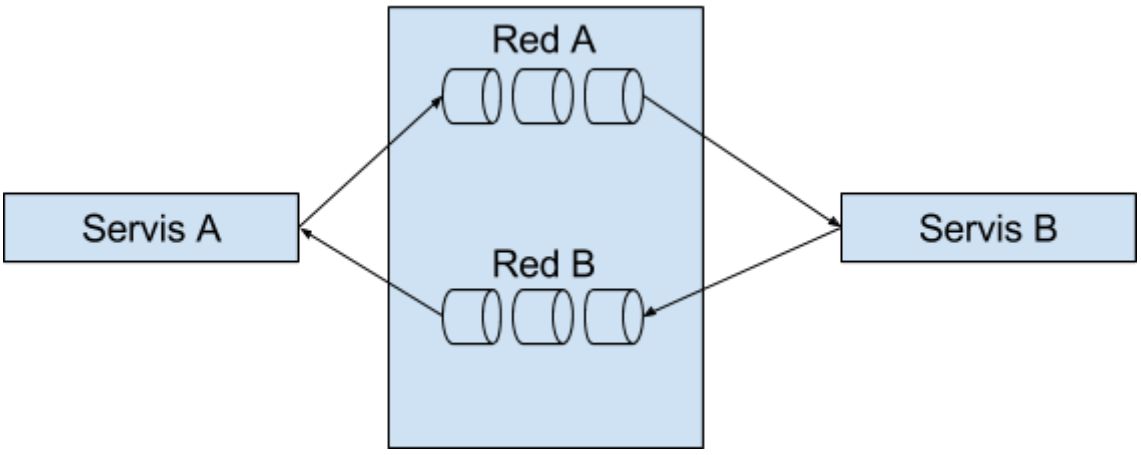
2.2.3 Obrazac asinhronih redova

Problem

Pretpostavimo da postoje dva servisa, takvi da jedan od njih, servis A, proizvodi poruku koju drugi servis, servis B, obrađuje i vraća odgovor. U takvom sistemu problemi nastaju kada je servisu B potrebno više vremena da poruku obradi i vrati odgovor. Ukoliko servis A čeka sve vreme na odgovor on zauzima svoje resurse i stoji čekajući, iako bi mogao da nastavi da obrađuje druge zahteve. Da bi se resursi servisa A oslobodili, dovoljno je da se u komunikaciji između ta dva servisa uvede red poruka, tako da servis A sve svoje poruke servisu B šalje u Red A. Servis B obrađuje poruke iz Reda A i odgovare šalje u Red B koji sluša servis A i tako se komunikacija nastavlja.

Rešenje

Dva servisa koja komuniciraju posredstvom para redova, svaki za zahteve jednog servisa i odgovore drugog servisa, naziva se obrazac asinhronih redova (slika 2)



Slika 2: Obrazac Asinhronih redova

Ukoliko servis A ima zadatak koji treba obaviti, on poruku šalje u Red A, i tako sve poruke koje je neophodno obraditi. Servis B uzima poruke, jednu po jednu iz Reda A i obrađuje. Kada obradi poruku, svoje odgovore šalje u Red B, koje Servis A uzima i obrađuje kada ima dovoljno resursa.

Redove je moguće implementirati na više načina. Pored podrške samih jezika za redove, poput *Java Queue*-a, postoje i gotovi sistemi za redove. Svakako servise poput ActiveMQ, ZeroMQ ili RabbitMQ je bilo moguće koristiti i u sistemima koji nisu u oblaku, ali i u onima koji jesu. Oni nude apstrakciju redova, koje korisniku nude alate da jednostavno deklarišu red i zatim uspostave komunikaciju izmedju proizvođača i korisnika poruka. Pored njih, većina poslužioca računarstva u oblaku nudi svoja rešenja za problem redova.

Tako *Amazon*, odnosno *AWS*, nudi svoj *Amazon Simple Queue Service,* iliti *Amazon SQS*. On predstavlja distribuiran sistem za razmenu poruka preko redova[14]. Amazon garantuje visoku skalabilnost i dostupnost, tako da ukoliko se koristi oni garantuju da će svaka poruka biti dostavljena makar jednom. Ovo znači da je moguće da jedna poruka bude dostavljena više puta, i zato sistemi koji koriste *Amazon SQS* moraju biti otporni na situaciju dupliciranih poruka. Ukoliko se koriste ovi redovi, oni omogućavaju da se jednostavnim skupom pravila dostigne laka skalabilnost, tako što se povećava sama veličina sistema redova. Pored manuelnog podešavanja i kreiranja redova, sve operacije nad istim moguće je vršiti i kroz aplikacije, koristeći *Amazon*-ov SDK. Slično rešenje nudi i Microsoft u svom oblaku *Azure*. Njihov sistem se naziva *Azure Service Bus Messaging*, i pruža praktično isti skup servisa kao i *Amazon*[15].

Google nudi rešenje koje se naziva *Google Pub-Sub*, koje je slično prethodnim navedenim ali nudi i dosta više[16]. Naime pored samog baratanja porukama, njihovim slanjem i primanjem, *Pub-Sub* nudi i opcije Protoka (*Streaming)*, koje pojednostavljuju prebacivanje velikih poruka. Odnosno, velike poruke je moguće obradjivati dok se dohvataju iz reda, za razliku od drugih sistema gde je neophodno prvo uzeti poruku, a zatim je obraditi. Takođe,

sistem je tako kreiran da ga je lako implementirati na različitim platformama, od mobilinh do desktop i poslovnih sistema.

Razlike između njih su jako male, međutim prednost *Amazon*-a je grupna obrada poruka. Kod *Amazon*-a je moguće izbrisati ili poslati više poruka odjednom, dok kod drugih takva opcija ne postoji. *Azure*-ova prednost se nalazi pre svega u mogućnosti obrade poruka kada se one nalaze na samom sistemu redova, bez uzimanja poruke iz reda. Kod njihovog sistema je moguće pročitati poruku ili je izmeniti dok se nalazi na samom redu.

Posledice

Uvođenje redova u sistem dodaje dosta fleksibilnosti. Kada jedan od servisa ne radi, drugi servisi mogu da nastave da rade normalno. Poruke će se gomilati u redovima servisa koji trenutno ne radi, ali čim se servis oporavi i nastavi sa izvršavanjem, uzimaće poruke i obrađivati. Na taj način ne gubi se nijedna poruka, a ceo sistem može da radi čak i ako neki od servisa ima problema. Komponente sistema postaju međusobno manje zavisne, i sam razvoj i održavanje servisa je jednostavniji.

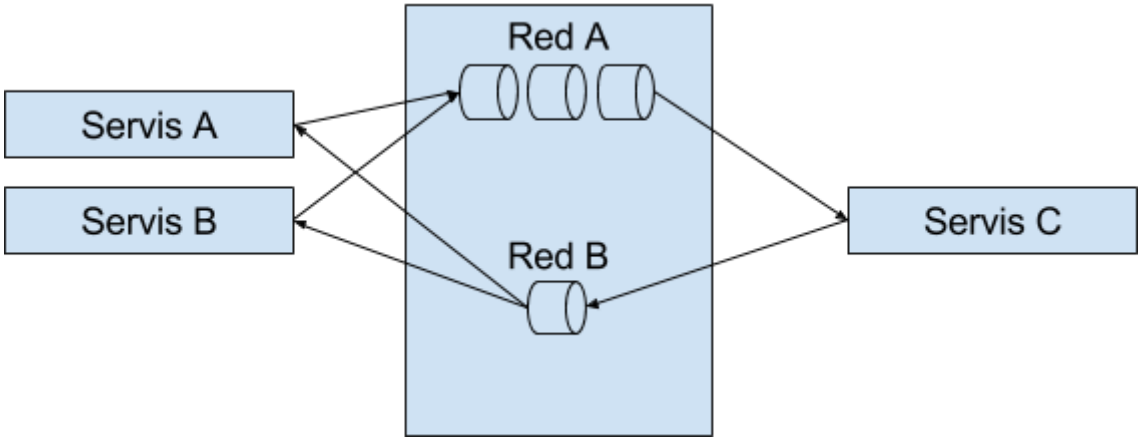
Glavna mana ovih obrazaca leži u činjenici da nekada ne možete da garantujete da će poruka stići do servisa kome je namenjena. Poruku je moguće pogrešno proslediti usred greške u kodu ili loše mrežne komunikacije koja može dovesti do gubljenja poruke. Većinu ovih problema preuzeli su poslužioci računarstva u oblaku sa svojim rešenjima. Oni brinu da servisi budu uvek dostupni, da lako skaliraju i da uvek isporuče poruku željenom servisu. Sam servisi koji koristi redove mora biti otporan na nedostupnost sistema redova. Odnosno ukoliko je red nedostupan neophodno je da sistem bude u stanju da pokušava više puta da pošalje ili primi poruke.

2.2.3.1 Obrazac obrade u serijama

Problem

Komunikacija preko redova donosi mnoge prednosti. Pre svega, ta prednost se ogleda u otpornosti sistema na ispadanje nekih njegovih delova. Posredstvom redova moguće je i otkloniti čekanje na odgovor sistema kojem se šalju zahtevi. Međutim kod ovakvih sistema često može doći do nagomilavanja poruka. Kako svaki servis ima određeni kapacitet poruka

koje može da obradi u nekom periodu, ukoliko je broj poruka veći od kapaciteta dolazi do zagušenja. U nekim slučajevima moguće je podići kapacitet servisa tako što se pojačaju serveri na kojima se servis nalazi, ali to zahteva dodatne troškove i nije uvek izvodljivo.



Slika 3: Obrazac obrade u serijama

Neka Servisi A i B proizvode poruke koje obrađuje Servis C, i neka se poruke šalju u Rad A odakle ih Servis C čita, kao na slici 3. Problem nastaje kada oba servisa proizvode poruke velikom brzinom, a Servis C ih ne obrađuje dovoljno brzo. Ukoliko je broj poruka koje proizvode servisi A i B veći od broja poruka koje servis C može da obradi dolazi do gomilanja poruka u Redu A i ceo sistem postaje zagušen.

Rešenje

Jedan od načina rešavanja ovog problem jeste pisanje Servisa C tako da ne obrađuje jednu po jednu poruku već da to čini u serijama. S obzirom da se veći deo vremena troši na uzimanje poruke iz reda i obrađivanje jedne po jedne poruke na identičan način, ukoliko se obrađuje više poruka istovremeno i sam sistem će imati veći kapacitet.

Pretpostavimo da Servisi A i B šalju podatke o prilivima na neki račun. Tada Servis C radi na način kao u primeru 2.

```
public void obradiPrihod(){
  while (true) {
    Prihod prihod = new Prihod();
    //dohvati sledecu poruku iz redA
    prihod = dohvatiSledecuPorukuIzReda(redA);
    obradiPrihod(prihod.getIznos(), prihod.getRacun());
  }
}
```

Primer 2: obrada pojedinačne poruke

Ukoliko `obradiPrihod()` traje *n*, za obradu svih *m* poruka iz reda potrebno je *m*n*. Međutim, kako se u ovakvim sistemima često dešava da se poruke odnose na isti događaj, u ovom slučaju dodavanje na isti račun, moguće je skinuti više poruka i obraditi ih zajedno. U primeru 3 obrada se vrši u serijama.

```
public void obradiPrihod(){
  //broj poruka koje se uzima iz reda
  int BROJ_PORUKA = 100;

  while (true) {
   Prihod[] prihodi = new Prihod[BROJ_PORUKA];
   //dohvati BROJ_PORUKA iz redA
   try {
    otvoriTransakciju();
    prihodi = dohvatiPorukeSaReda(redA, BROJ_PORUKA);
    //grupisi poruke po računu na koji se odnosi
    Map<String, Object> iznosiPoRacunu = new HashMap<>();
    for (prihod : prihodi) {
      //proverava da li postoji unos za taj kljuc, i dodaje vrednost
      //ukoliko postoji, odnosno postavlja ukoliko ne postoji
      Float iznos = iznosPoRacunu.containsKey(prihod.getRacun()) ?
```

```
iznosPoRacunu.getValue(prihod.getRacun()) + prihod.getIznos() :
prihod.getIznos();
      iznosPoRacunu.set(prihod.getRacun(), iznos);
    }
    //za svaki racun odraditi transakciju
    for (prihodPoRacunu : iznosiPoRacunu.entrySet()) {
      //obrada cele serije promena odjednom
      obradiPrihod(prihodPoRacunu.getValue(), prihod.getKey());
    }
     zatvoriTransakciju();
   }
   catch(Exception e) {
     vratiPoruke(prihodi);//u slucaju greske vrati poruke u red
     vratiTransakciju(); //vrati transakciju
   }
  }
}
```

Primer 3: Obrada u serijama

Posledice

Uzimanjem većeg broja poruka dobija se višestruka ušteda. Kao prvo, neophodna je samo jedna veza ka servisu ili redu gde se poruke čuvaju. Zatim, umesto da se obrađuju poruke jedna po jedna, uzimanjem većeg broja poruka moguće je grupisati ih po entitetima na koje se odnose. Umesto da postoji obrada po poruci, obrada je sada po samom objektu po kome se grupiše, u ovom slučaju po računu. U najgorem slučaju, ukoliko se svaka poruka odnosi na različit objekat, trajanje obrade poruka je isto kao i kada nemamo obrazac obrade u serijama, $m*n$. Međutim, u najboljem slučaju, ukoliko su sve poruke vezane za isti objekat, tada obrada svih poruka je n, jer se u jednom koraku obrađuju sve.

Odnosno, bez obrade poruka u serijama, trajanje obrade je $m*n$, gde je m broj poruka a n trajanje obrade jedne poruke. Sa implementiranim obrascem poruka u serijama, trajanje obrade je $l*n$, gde je sada l broj objekata koje je neophodno obraditi. S tim što je l uvek manje od n.

Svaki put kada se poruke uzmu sa reda, neophodno ih je sortirati. Ukoliko sortiranje zahteva više resursa nego sama obrada poruke, potrebno je dobro proceniti da li ovaj obrazac

donosi dovoljno prednosti. Takođe, veličina ovih poruka ima veliku ulogu, jer ukoliko su broj i veličina poruka veliki, može doći do popunjavanja memorije prilikom izvršavanja i tako dovesti do neželjenih posledica poput pucanja programa, zaglavljivanja i slično.

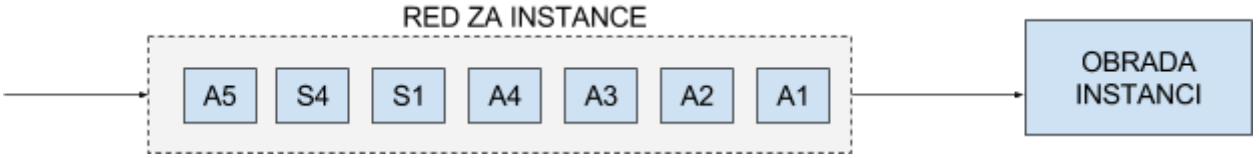
Treba imati u vidu i da sistem uzimanja poruka treba napisati tako da nikada ne čeka da se nakupi dovoljno veliki broj poruka. Neophodno je da uvek uzima onoliko poruka koliko ih ima u redu. Ukoliko se čeka da se skupi veći broj poruka, dolazi do odlaganja obrade pojedinih poruka, što može uneti dodatno odlaganje prilikom obrade poruka.

2.2.3.2 Obrazac redova prioriteta

Problem

Kod korišćenja redova, sve poruke se šalju na jedan red i obrađuju se redom kojim su pristigle. Međutim, u stvarnim situacijama nemaju sve poruke istu težinu, odnosno vreme čekanja neke poruke je skuplje od čekanja neke druge. Poruke veće težine moraju biti uslužene sa što manjim čekanjem, čak iako druge poruke trpe zbog toga. Zato, generalno, redovi koji rade u režimu ”prvi na ulazu, prvi na izlazu” , odnosno *FIFO,* znaju da stvore probleme.

Pretpostavimo da postoji jedan red na koji dolaze poruke za pokretanjenje i zaustavljanje instanci (virtuelnih servera), kao na slici 4.



Slika 4: Red sa porukama različitog prioriteta

U redu se nalaze poruke za podizanje instanci, A1, A2, A3, A4 i A5, ali i poruke da se zaustave instance 1 i 4, u porukama S1 i S4.

Ovakav redosled poruka je moguć kada korisnik pokrene akciju pokretanja instance, međutim odmah nakon pokretanja korisnik se predomisli i otkaže taj zahtev, odnosno zaustavi podizanje instance. S obzirom da zahtev za instancom 1, A1, nikada nije obrađen, očekivano je da se ta instanca nikada ne podigne, odnosno zatraži od poslužioca oblaka.

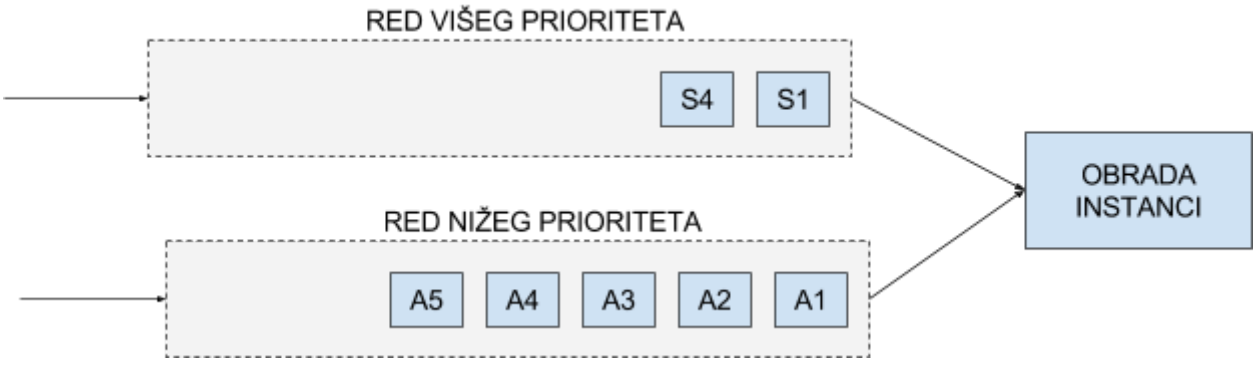
Međutim, kako je red za instance popunjen, i obrada instanci obrađuje jedan po jedan događaj, prvo će se obraditi događaj A1, a tek nakon što se obrade sledeća 3 zahteva, stićiće događaj za zaustavljanje instance.

Na primeru instanci možemo videti da trošak u obradi poruka nije zanemarljiv. Kako je instanca već pokrenuta, ona će biti zaustavljena, ali i naplaćena. U slučaju *Amazon*-a[17], instanca se naplaćuje po satu, pa za par sekundi kašnjenja u obradi poruke, instanca će biti naplaćena kao da je korišćena ceo sat. Kod *Microsoft Azure*-a[18] je dosta povoljnije, jer se instanca naplaćuje po minutu. Kod *Google Cloud*-a[19], situacija je slična, jer je naplata po iskorišćenom minutu, ali se prvih 10 minuta uvek naplaćuje. Kašnjenje u obradi poruka može uvesti dodatne troškove ili nekonzistentnost u sistemu.

Rešenje

S obzirom da poruke koje se obrađuju se razlikuju po prioritetu, nema smisla obrađivati ih na isiti način. Rešenje ovog problema se zasniva na mogućnošću uvođenja dva različita reda, iz kojih bi se poruke obrađivale različitim prioritetom. Poruke se razdvajaju na dva reda, tako da obrađivač uvek prvo čita poruke iz reda višeg prioriteta, pa tek nakon toga i poruke nižeg prioriteta.

U datom primeru možemo razdvojiti redove na red koji bi čuvao poruke o zaustavljanju instanci, i red koji bi imao poruke vezane za podizanje novih. Na slici 5 možemo videti razdvojene redove u primeru sa instancama.

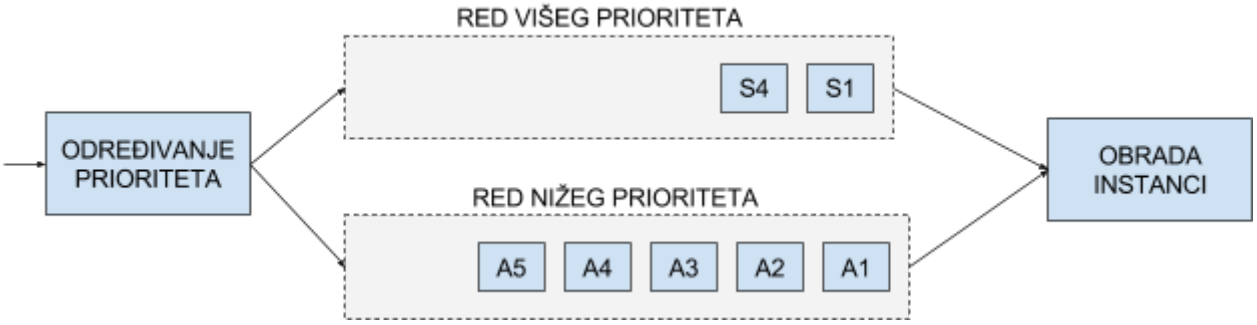


Slika 5: Razdvojeni redovi prioriteta

Obrada instanci u tom slučaju uvek prvo kreće od poruka iz reda višeg prioriteta, pa tek kada je on prazan sistem prolazi kroz red nižeg prioriteta.

U primeru, obrada instanci prvo obrađuje zahtev za zaustavljanje instanci 1 i 4, S1 i S4. Red višeg prioriteta nakon toga ostaje prazan, servis prelazi na red nižeg prioriteta, i dobija poruku o pokretanju instance 1, A1. Kako je servis već obradio suprotan zahtev, odnosno zahtev za zaustavljanjem te instance, ova poruka se zanemaruje i prelazi se na sledeću poruku.

Naravno redovi, odnosno prioriteti, mogu biti definisani na različite načine, zavisno od samog sistema i zahteva proizvoda. Nekada je razdvajanje potrebno raditi po kriterijumima kao što su važnost samih klijenata, veličina i zahtevnost samih poruka i slično. Kod kompleksnije logike razdvajanja, dodavanje servisa koji razdvaja poruke daje veće mogućnosti i dobija na brzini, kao na slici 6.



Slika 6: Redovi prioriteta sa određivačem prioriteta

Posledice

Problemi koji mogu da nastanu su pre svega vezani za određivanje prioriteta. Ukoliko previše poruka ide u red višeg prioriteta, preti opasnost da se poruke nižeg prioriteta nikada ne obrađuju, a samim tim ovaj red može postati zagušen ogromnim brojem poruka.

Taj problem se može izbeći ako postoji jedan korisnik reda (onaj koji čita poruke iz reda) ukoliko se obrađuju poruke višeg prioriteta sve dok se ne ispuni jedan od dva uslova:

- nema više poruka višeg prioriteta

- obrađeno je više od definisane maksimalne količine poruka višeg prioriteta koje je moguće obraditi pre provere reda nižeg prioriteta (*nRVP)*.

Kada se ispuni jedan od dva uslova prelazi se na obradu reda nižeg prioriteta. On se ponaša isto, ali su parametri drugačije definisani. Maksimalni broj poruka koji se uzima iz reda nižeg prioriteta (*nRNP*) je uvek manji nego na redu višeg prioriteta. Na taj način garantujemo da će na svakih *nRVP*, broj obrađenih poruka nižeg prioriteta uvek biti onoliko koliko ih ima ili *nRNP*, gde je *nRNP* << *nRVP*.

Ukoliko više korisnika reda obrađuju poruke, tada je implementacija reda po korisniku poželjna, ali mora se voditi računa da korisnik koji obrađuje red višeg prioriteta ima i više resursa, kako bi radio brže, kao i da logika same aplikacije bude takva da bude u skladu sa konkurentnom obradom poruka. Takođe, kombinacija redova prioriteta i obrade u serijama, potencijalno daje odlične rezultate.

2.2.4 Obrazac ponavljanja

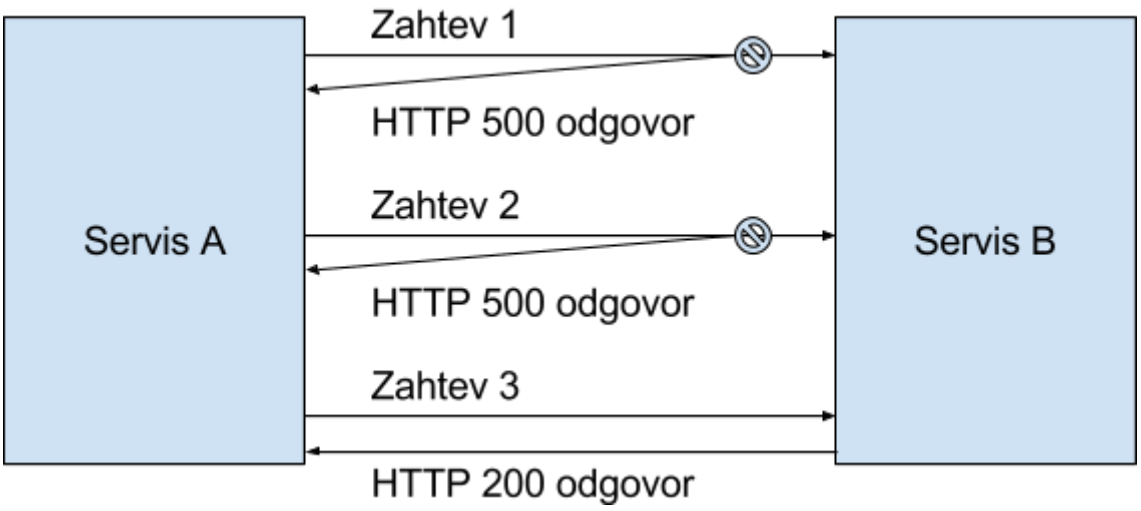
Problem

Kada se sistem sastoji iz većeg broja komponenti, ili ukoliko komunicira sa spoljnim servisima, uvek postoji mogućnost da neka od komponenti ne odgovara na zahteve. Ukoliko servis sa kojim pokušavamo da komuniciramo nije dostupan operacija se može proglasiti neuspelom. Međutim kako cena neuspeha može biti veoma visoka, uvek je neophodno uraditi što je više moguće da do toga ne dođe. Jedan od najjednostavnijih načina za povećanje stabilnosti sistema je uvođenje obrasca ponavljanja.

Rešenje

Aplikacija koja komunicira sa nekom drugom izložena je problemima poput gubitka mreže, primanje loših paketa, isticanja vremena posle koga se očekuje odgovor i slično. Takve greške se najčešće otklanjaju tako što se ponovi operacija. Priroda takvih problema je trenutna, i oni ne zahtevaju intervenciju korisnika kako bi bili uklonjeni.

Ukoliko servis A šalje zahtev servisu B, i ako je servis B trenutno nedostupan ili mu je neophodno dosta dugo da odgovori, servis A ponavljanjem zahteva se može uspešno oporaviti od problema u velikom broju slučajeva. Na slici 7, možemo da vidimo primer kada servis A pokušava dva puta da pošalje poruku servisu B pre nego što zahtev postane uspešan. Prvi i drugi zahtev su dobili odgovor da nešto trenutno nije u redu, odnosno da servis ne može da obradi taj zahtev. Nakon dva ponavljanja zahtev je uspešno prosleđen i odgovor dobijen.



Slika 7: Obrazac ponavaljanja

Nestalne, odnosno privremene, greške svaki servis mora da obrađuje tako što ih prvo analizira, i ukoliko ustanovi da je verovatnoća ponavljanja greške mala, neophodno je da ponovo pokuša isti zahtev. Tipični primeri grešaka koje servis treba da proba jesu one iz serije 500 HTTP grešaka. Međutim, greške poput onih iz serije 400 HTTP grešaka nema potrebe pokušavati ponovo, i zato je obrada svake greške neophodna pre ponovnog pokušaja. Osnovni način implementacije možemo videti na primeru 4.

```
public HttpResponse obaviHTTPZahtevSaPonavljanjem(
                    HttpRequestBase zahtev) throws IOException
  {
    //koliko puta ponoviti HTTP zahtev
    static Integer BROJ_PONAVLJANJA = 5;
    IOException poslednjaGreska = null;
    Integer brojPokusaja = 0;
    while (true){
      try {
        HttpResponse odgovor = httpClient.execute(httpZahtev);
        int statusCode = response.getStatusLine().getStatusCode();
       //ukoliko status kod koji se vrati ne treba pokusavati
       //ponovo prekinuti operaciju i vratit poslednji odgovor
        if (!daLiJePonovljiv(statusCode)) {
          return response;
```

```
        }
      } catch (IOException e) {
        httpRequest.abort();
        poslednjaGreska = e;
      }
      finally{
        //povecati broj pokusaja
        brojPokusaja++;
        if (brojPokusaja > BROJ_PONAVLJANJA){
          throw poslednjaGreska;
        }
      }
    }
    throw poslednjaGreska;
  }
```

Primer 4: Osnovni obrazac ponavljanja

Naravno, ovaj osnovi tip obrasca moguće je unaprediti na mnogo načina. Osnovna mana ovog obrasca je što aplikacija pokušava odmah sa novim zahtevom čim se prethodni završi. To može dovesti do problema, odnosno ne mora dati željene rezultate, jer se jako brzo pokuša sa maksimalnim brojem zahteva i ne da se prilika sistemu da se oporavi. Takođe, ukoliko se šalje previše zahteva ka nekom servisu, servis iz odbrambenih razloga može odbaciti sve sledeće zahteve misleći da zahteve šalje zlonamerni napadač na servis.

Jedna od mogućnosti rešavanje ovih problema jeste dodavanje perioda odlaganja između dva zahteva, tako da nakon svakog neuspešnog zahteva sačeka se određeno vreme pre sledećeg pokušaja, kao u primeru 5. Tako između svaka dva zahteva daje se prilika da se servis, ili mreža oporave i na taj način povećava se mogućnost da zahtev bude uspešno obrađen.

```
public HttpResponse obaviHTTPZahtevSaPonavljanjem(HttpRequestBase
                                httpZahtev) throws IOException {
    //koliko puta ponoviti HTTP zahtev
```

```
    static Integer BROJ_PONAVLJANJA = 5;
    IOException poslednjaGreska = null;
    Integer brojPokusaja = 0;
    long vremeSpavanjaMilisec = 5000;
    while (true){
      try {
        …
      } catch (IOException e) {
        …
      }
      finally{
        //povecati broj pokusaja
        brojPokusaja++;
        if (brojPokusaja > BROJ_PONAVLJANJA){
          throw poslednjaGreska;
        }
        //uspavaj thread kako bi imalo vremena da se oporavi servis
        Thread.sleep(vremeSpavanjaMilisec);
      }
    }
    throw poslednjaGreska;
  }
```

Primer 5: Obrazac ponavljanja sa odlaganjem

Međutim i ovakav sistem je moguće unaprediti tako što, umesto da se uvek čeka fiksni period, potrebno je uvesti eksponencijalno vreme čekanja. Na taj način, ukoliko je došlo do greške, pokušava se što pre, ali ukoliko se greška pojavljuje više puta zaredom, povećavamo vreme čekanja i dajemo priliku servisu da se oporavi.

Prethodni kôd sada izgleda kao na primeru 6.

```
public HttpResponse obaviHTTPZahtevSaPonavljanjem(HttpRequestBase
                                httpZahtev) throws IOException {
    //koliko puta ponoviti HTTP zahtev
    static Integer BROJ_PONAVLJANJA = 5;
    IOException poslednjaGreska = null;
    Integer brojPokusaja = 0;
    long vremeSpavanjaMilisec = 5000;
    while (true){
      try {
        …
      } catch (IOException e) {
        …
      }
      finally{
        //povecati broj pokusaja
        brojPokusaja++;
        if (brojPokusaja > BROJ_PONAVLJANJA){
          throw poslednjaGreska;
        }
        //uspavaj thread kako bi imalo vremena da se oporavi servis
        Thread.sleep(Math.round(
                        vremeSpavanjaMilisec*Math.pow(2,attempt)));
      }
    }
    throw poslednjaGreska;
  }
```

Primer 6: Obrazac ponavljanja sa eksponencijalnim vremenom čekanja

Posedice

Ovaj obrazac je potrebno implementirati samo kada odgovara zahtevima aplikacije. Nekada je cena ponovnog pokušavanja velika i priroda sistema je takva da je poželjno da, ukoliko je nemoguće izvršiti zahtev, što pre proglasimo operaciju neuspelom i pustimo sistem da radi dalje. Takođe, ukoliko sistem pokušava prevelik broj puta u malom vremenskom

intervalu, sistem koji je ionako bio pod velikim opterećenjem biva optrećen sa još većim brojem zahteva i time se rizikuje stabilnost celog sistema.

Kod ovakvih sistema treba obratiti pažnju na to da poruke koje sistem šalje mogu biti primljene od strane servera za obradu, iako odgovor nije dobijen usled greške na klijentskoj strani. U tom slučaju neophodno je imati u vidu mogućnost da se ista poruka prosledi dva puta na obradu. Sistem mora biti otporan na takve događaje, i takve poruke ne smeju dovesti do nekonzistentnog stanja.

Skupljanje događaja prilikom ponavljanja može doneti značajne informacije za dalje unapređenje sistema. Ukoliko se beleži svako ponavljanje nekog zahteva, onda možemo imati jasnu sliku koliko se često javlja određen tip greške, posle koliko ponavljanja se zahtev uspešno izvrši i slično. Na osnovu tih podatka stvara se jasna slika gde i na koji način se može unaprediti sistem.

Glavna prednost ponavaljanja zahteva je kod infrastrukturnih grešaka kao što su privremeno ispadanje servisa, mrežni problemi, trenutni nalet velikog broja zahteva i slično. U takvim slučajevima ponavljanje može eliminisati prekide u rada sistema.

2.2.5 Obrazac strujnog kola

Problem

Sa velikim brojem servisa, raste i mogućnost da u nekom trenutku jedan od servisa prestane da radi i ispadne iz sistema. Iako se često te greške brzo isprave, i dovoljno je samo ponoviti zahtev kao u obrascu ponavljanja, ponekad to nije dovoljno. Problem može biti dosta dublji i samom servisu je potrebno više vremena da se oporavi. Ti problemi mogu biti zbog gubljenja veze na duži period do potpunog prestanka rada servisa. U takvim slučajevima, uglavnom, je najbolje prihvatiti činjenicu da taj servis ne radi i obrađivati greške na adekvatan način.

U slučajevima, kada servis kome se upućuju zahtevi trpi veliki saobraćaj, odnosno ima veliki broj zahteva, servis se brani tako što odbacuje zahteve koje ne može da obradi jer je iskoristio sve svoje kapacitete. Ovo se može desiti i ukoliko je servis iskoristio sve svoje resurse, poput procesorske snage ili iskorišćenja dostupne memorije. U tim slučajevima zahtevi se odbacuju dok se ne oslobode resursi i tek onda se prihvataju novi zahtevi.

Obrazac asinhronih redova ovo može rešiti ali samo kada je moguće da obrada zahteva bude asinhrona, ukoliko to nije moguće i zahtev mora biti odmah obrađen i odgovor vraćen servisu, onda taj obrazac nije od pomoći.

Rešenje

Obrazac strujnog kola može da spreči servis da često pokušava da obavi operaciju koja se najverovatnije ne može izvršiti. Na taj način se sistem sprečava da troši previše resursa na pokušavanje obrade zahteva, iako zahtev u tom trenutku najčešće i nije moguće obraditi. Ovakve greške se dešavaju kada je baza nedostupna i tada umesto slanja zahteva ka bazi i obrađivanja greške, što bi zahtevalo dosta vremena i resursa, obrazac strujnog kola ustanovljava da baza ne odgovara na zahteve i sledeće zahteve ne obrađuje. Odnosno ne šalje zahteve ka bazi, već samo pokreće proceduru obrade nedostupnosti baze.

Aplikacije računarstva u oblaku vrlo često zahtevaju neki resurs, poput instance, od samog poslužioca. Poslužioci koriste sisteme koji ne dozvoljavaju previše zahteva od strane jednog korisnika. Takav slučaj je sa limitom koji određuje koliko se instanci određenog tipa može podići u datom vremenu. Ukoliko se nastavi sa slanjem velikog broja zahteva ka poslužiocu, neki od njih, poput *Amazon-a* i *Azure*-a, odbijaju sve dalje zahteve i na taj način kažnjavaju korisnika zbog prevelikog broja zahteva, koje svakako nije bilo moguće obraditi. U takvim situacijama je najbolje ne slati zahteve, već uvesti obrazac strujnog kola kod prvih naznaka da se zahtevi ne mogu opslužiti.

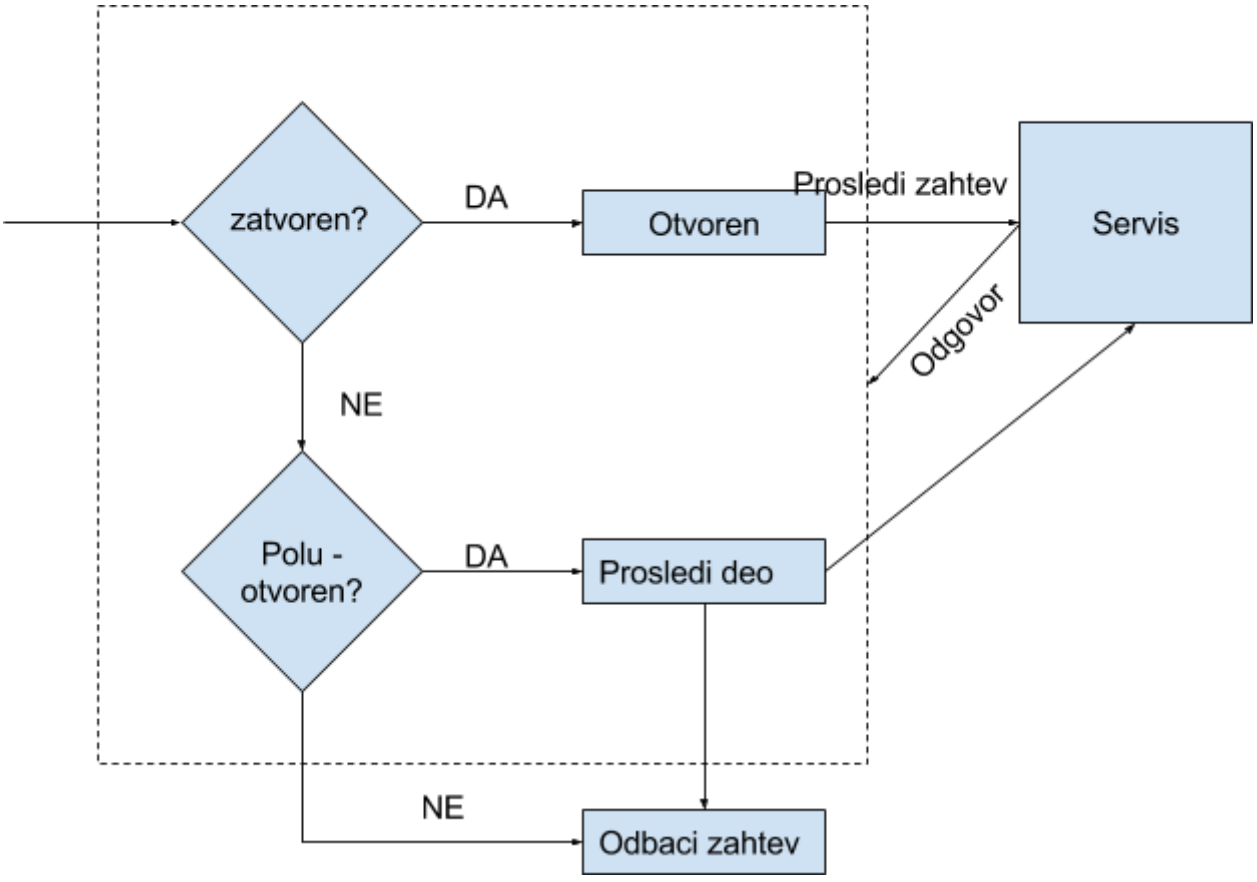
Obrazac strujnog kola radi tako što svi zahtevi kod kojih postoji naznaka da mogu biti neobrađeni prolaze kroz obrazac, a on na osnovu metrika određuje da li će proslediti zahtev ili ne.

Ovaj obrazac možemo posmatrati kao mašinu stanja u sa tri stanja:

- **Otvoren**: ili isključen, po analogiji na strujno kolo. Ukoliko je u ovom stanju, obrazac će proglasiti neuspešnim sve zahteve koje dobije. Neće ni pokušavati da ih obradi, već će odmah vratiti grešku
- **Zatvoren**: kada je strujno kolo zatvoreno, tada struja neometano teče. Tako i u obrascu strujnog kola, ukoliko je u zatvorenom stanju svi zahtevi se prosleđuju na izvršavanje. Pri obradi svakog zahteva, aplikacija vodi računa da li je došlo do greške. Ukoliko se pojavila greška povećava se brojač. Kada broj grešaka prevaziđe zadati broj u datom periodu, kolo se otvara i svi zahtevi se odbacuju. Kada prođe period definisano kao vreme zatvaranja, odnosno koliko dugo se automatski odbacuju sve greške, stanje se menja u poluzatvoreno.

- **Poluzatvoreno**: U poluzatvorenom stanju prosleđuje se samo deo zahteva. Ukoliko neki zahtev ponovo dobije grešku, stanje se ponovo vraća u zatvoreno i merač vremena se ponovo pokreće. Na taj način se produžava period koje je dato servisu da se oporavi. Ukoliko se zahtevi uspešno izvrše, proglašava se da je servis ponovo dostupan, svi brojači se anuliraju i sistem prelazi u otvoreno stanje. Razlog zbog koga se u poluzatvorenom stanju ne prosleđuju svi zahtevi, već samo neki, jeste da se ne zaguši servis koji vraća grešku, i na taj način se obezbeđuje vreme neophodno za oporavak.

Svaki zahtev se obrađuje kao na slici 8. Kada zahtev stigne proverava se da li je strujno kolo zatvoren ili ne. Ukoliko je zatvoreno, odnosno sistem radi kako treba, zahtev se prosleđuje na izvršavanje. Ukoliko kolo nije zatvoreno, onda se proverava da li je u poluotvorenom stanju. U poluotvorenom stanju se određen broj zahteva propušta, a ostali odbacuju, dok u otvorenom se svi odbacuju.



Slika 8: Obrazac prekidača

Svaki odgovor i svaki zahtev menjaju stanje u kom se nalazi kolo. Potrebno je definisati pravila kada stanje prelazi iz jednog u drugo stanje. Pravila koja treba definisati, za prelazak iz stanja u stanje su:

- Iz zatvorenog u poluotvoren: ukoliko u poslednjih *n* minuta je bilo *m* neuspešnih zahteva
- Iz otvorenog u poluotvoren: nakon isteka perioda *t* u otvorenom stanju, koje je definisano kao vreme koliko sistem ostaje u poluotvorenom stanju
- Iz poluotvorenog u otvoren: ukoliko je procenat neuspešnih zahteva veći od procenta uspešnih
- Iz poluotvorenog u zatvoren: ukoliko je procenat uspešnih zahteva veći od procenta neuspešnih

Posledice

Obrazac ponavljanja i obrazac strujnog kola su veoma slični, međutim oni imaju različitu svrhu. Dok obrazac ponavljanja služi za ponavlje zahteva više puta dok ne uspe, obrazac strujnog kola ne prosleđuje zahteve koji će verovatno pasti, štedeći vreme obrade i dajući priliku servisu da se oporavi. Ova dva obrasca je moguće koristiti zajedno, tako što bi se obrazac ponavljanja koristio u obrascu strujnog kola kod upućivanja zahteva. Međutim, treba biti oprezan da se obradi svaka greška koja se dobija, kako bi obrazac strujnog kola imao relevantno stanje.

Glavna opasnost kod korišćenja ovog obrasca je ukoliko se obrada grešaka obavlja u delu koda koji se bavi prosleđivanjem zahteva, odnosno u obrascu strujnog kola. Ukoliko se to desi, sam obrazac usporava sistem i njegova kompleksnost postaje veća. Razumevanje koda je teže i samo održavanje sistema je skuplje. Pored toga, s obzirom na dodatnu logiku koja se uvodi, moguće je drastično usporiti sistem. Greške se uvek moraju prosleđivati ka klijentu koji je prosledio zahteve. On je taj koji mora da postavi svoju strategiju kako reagovati na određene greške, a deo koda sa obrascem strujnog kola samo prosleđuje zahteve i odgovore.

2.2.6 Obrazac mikroservisa

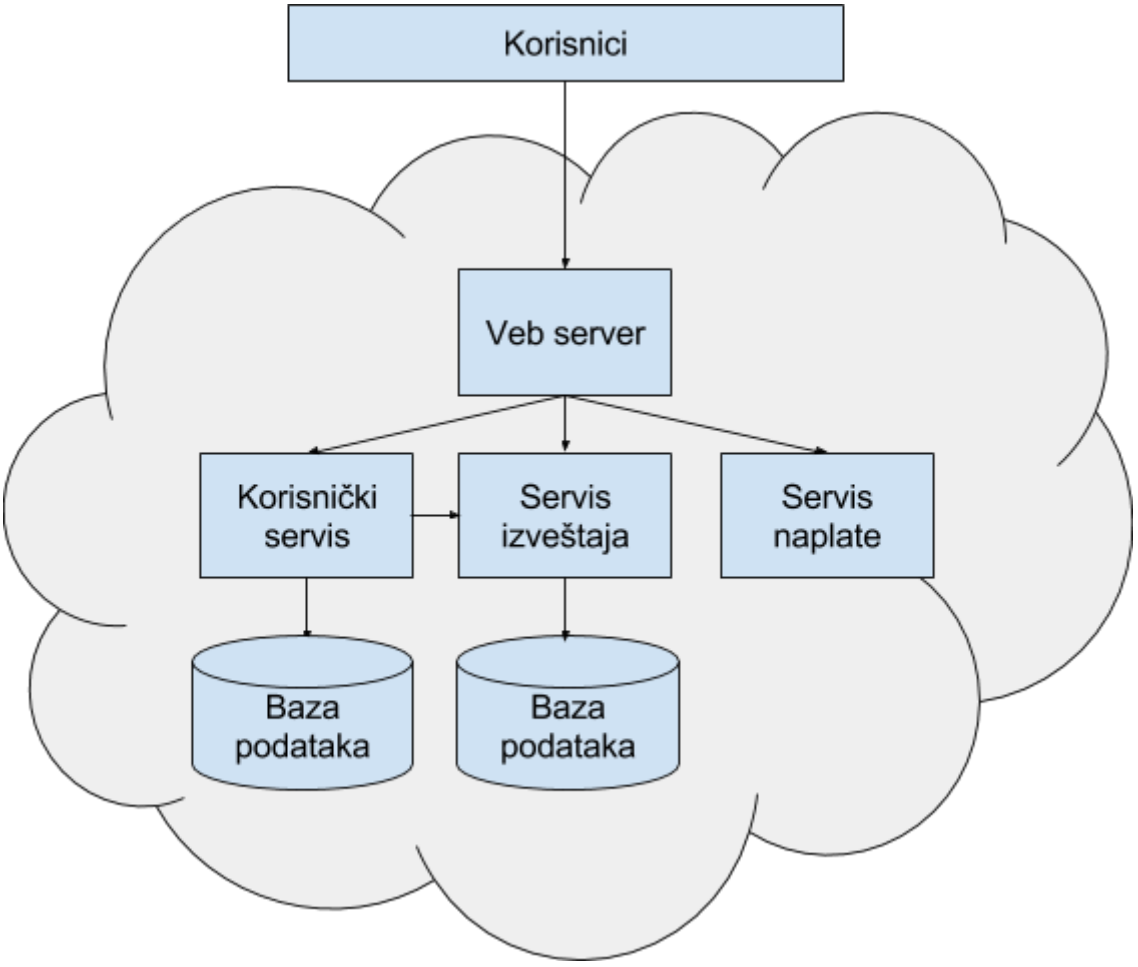
Problem

Prilikom razvoja sistema moguće je sistem postaviti na dva različita načina. Inicijalno najlakše je razmatrati jedan monolitni sistem, kod koga su sve funkcionalnosti zapravo deo jednog jedinstvenog servisa. Ovakav sistem je lak za početno razvijanje jer nije neophodno razmišljati kako komponente sistema međusobno komuniciraju, voditi računa o međusobnoj autentikaciji i autorizaciji. Tako se generalno izbegavaju problemi vezani za asinhroni sistem ili sistem kod koga je mrežna komunikacija jedan od bitnijih činioca. Međutim, takav sistem vrlo brzo postane ogroman i komplikovan. Održavanje takvog sistema zahteva puno resursa, dodavanje novih funkcionalnosti je jako komplikovano i sistem svojim rastom gubi na performansama. Alternativa, koja se često koristi u oblaku, jeste obrazac mikroservisa.

Rešenje

Mikroservis obrazac predstvalja skup slobodno povezanih servisa koji međusobno sarađuju, čineći jedinstven sistem. Svaki servis ima svoj domen u kome deluje, i rešava samo određeni skup problema i funkcionalnosti koje pruža čitav servis.

Na primer, sistem može imati servise koji se bave samo privilegijama pristupa, naplatama i slično. Na slici 9 vidimo primer sistema koji se sastoji od mikroservisa. Veb servis komunicira sa ostalim servisima i bavi se celokupnom komunikaciom sa korisnicima. On za obavljanje zadataka koje ima može koristiti usluge ostalih servisa poput Korisničkog, Servisa naplate ili Servisa izveštaja. Ukoliko je potrebno, svaki od ovih servisa može imati svoju bazu. On funkcioniše nezavisno i ima svoj domen delovanja. Takođe, pruža niz funkcionalnosti drugim servisima koji tako mogu da komuniciraju sa njim, dobiju određene podatke ili pošalju određene zahteve.



Slika 9: Mikroservis arhitektura

Servisi međusobno komuniciraju preko nekog od sinhronih protokola poput HTTP, REST protokola ili asinhronog poput AMQP.

Svaki od servisa se koncipira na takav način da su što nezavisniji jedan od drugog. Svaki od servisa mora biti spreman na to da drugi servisi neće uvek biti dostupni i u skladu sa tim mora voditi računa šta i kako raditi u tim slučajevima.

Posledice

Kod obrasca mikroservisa veličina pojedinačnog servisa je drastično manja nego veličina jednog monolitnog sistema. Rasčlanjen sistem je zato jednostavnije razumeti, stoga je i proširivanje tima ljudi koji rade na projektujednostavnije. Sama aplikacija se lakše i brže pokreće, a i sam proces je jednostavniji. Naime, umesto da se ceo sistem pokreće odjednom, moguće je to odraditi na jednom po jednom servisu. Zbog toga se smanjuje trajanje nedostupnosti celog sistema, lakše je planirati pokretanje nove verzije sistema i puštanje u

produkciju, jer se pušta deo po deo, nezavisno jedan od drugog. Takođe, ovakav pristup omogućava lakšu promenu tehnologija korišćenih za neki od servisa. Promene su, uglavnom, izolovane samo na taj servis, a ne na ceo sistem. Zbog razdvojenosti delova sistema, lakše je otkriti u kom delu sistema su greške nastale. A i sami problemi su izolovani, tako da ukoliko dođe do curenja memorije, vrlo je lako ustanoviti u kom delu sistema je problem i fokusirati se na rešavanje problema.

Međutim, obrazac mikroservisa uvodi dodatne probleme održavanja. Sami programeri moraju da se izbore sa kompleksnošću sistema jer je neophodno razviti sistem za internu komunikaciju. Testiranje ovakvog sistema je dosta teže, i sami scenariji za testiranje su dosta kompleksiniji. Oni zahtevaju asinhronu komunikaciju različitih servisa. Međutim, u sistemima obično ne postoje distribuirane transkacije što dodatno usložnjava proces testiranja. Kod mikroservisa raste potreba za kompleksnim integracionim testovima koji će proveriti kako sistemi rade zajedno. Ovo ne umanjuje važnost i neophodnost jediničnih testova, jer oni svakako moraju testirati unutrašnju logiku svakog od sistema posebno.

Ovi sistemi donose dosta poboljšanja u kontekstu puštanja sistema u rad, ali kako je ceo sistem dosta komplikovaniji i sama procedura pokretanja je dodatno kompleksna. Prvo, različit skup tehnologija koji se koristi čini da sam proces postaje složeniji. Promene formata poruka koje se razmenjuju između servisa unose veliku kompleksnost, jer je neophodno da se isplanira trenutak kada se koji deo radi i pušta, kao i pokrivanje ivičnih slučaja zbog asinhronosti sistema.

Takođe ovakvi sistemi zahtevaju više memorija za izvršavanje za razliku od monolitnog servisa. Odnosno, svaki od servisa zahteva na primer svoj JVM, svoju virtuelnu mašinu, svoj veb server i slično.

2.2.7 Obrazac otkucaja srca

Problem

Nedostupnost servisa ili potpun prestanak rada nekog od servera je česta situacija. Od grešaka vezanih za mrežu ili sličnih grešaka moguće je odbraniti se nekim od obrazaca o kojima je bilo reči poput obrasca ponavljanja. Međutim, kada je servis „mrtav”, odnosno potpuno prestane da radi, u većini slučajeva potrebno je uraditi skup novih instrukcija, kako bi se servis oporavio, ili pokrenuo nov.

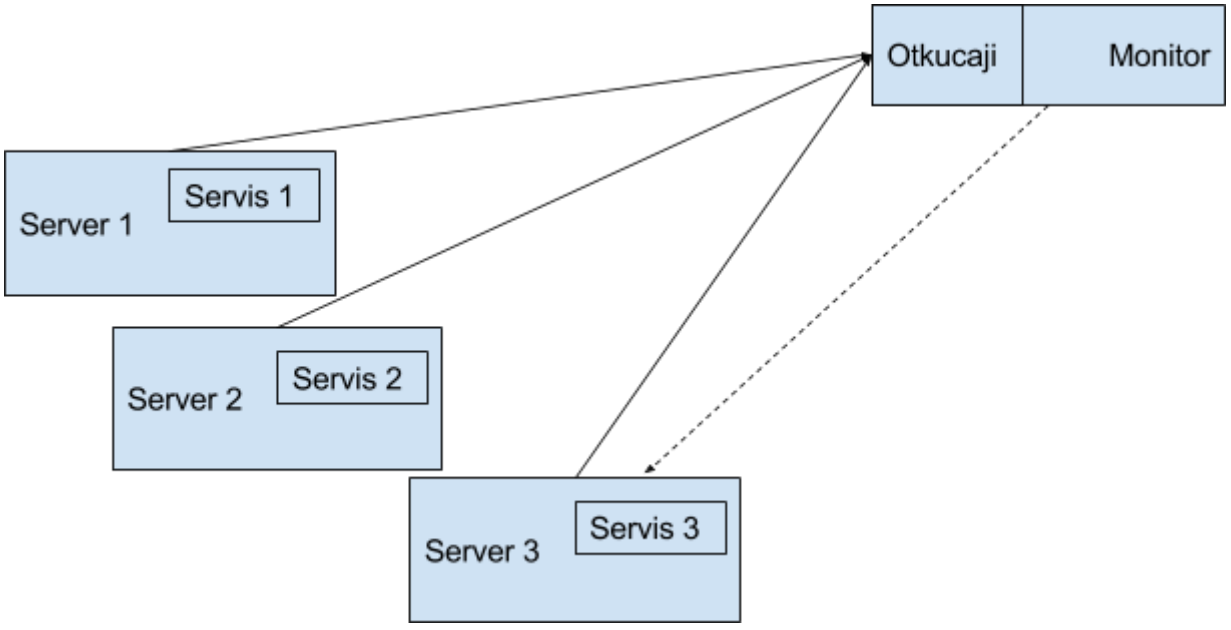
U oblaku je ovo ponašanje izraženije nego kod tradicionalnih sistema jer serveri mogu da prestanu da rade, i zbog događaja koji su van dometa inžinjera, na primer problemi ili održavanja na strani poslužioca oblaka. Ovi problemi su izraženiji i uvođenjem novih načina korišćenja instanci, poput instanci koje su dostupne samo određeno vreme.

Amazon je uveo način korišćenja instanci koje naziva *Spot*, i čija se cena određuje po sistemu koji je sličan berzanskom poslovanju. Svaki korisnik nudi koliko je spreman da plati za instancu, i ukoliko postoje dostupne instance po toj ceni dobija je na korišćenje. Međutim, u svakom trenutku, moguće je da neko ponudi više novca, i da preuzme instancu. *Google* nudi sličnu uslugu, u smislu instanci koje su jeftinije i mogu biti oduzete u bilo kom trenutku, ali sa razlikom što je cena fiksna, i do 80% jeftinija. Ove *Preemptible* instace zahtevaju isti način korišćenja i nadgledanja kao i *Amazon*-ove *Spot*.

U tim i sličnim slučajevima potrebno je otkriti da nešto nije u redu sa servisom, kako bi se pokrenula procedura oporavka, nevezano za to da li se radi o proceduri oporavljanja servisa, ili podizanja nove instance i ponovnog pokretanja servisa.

Rešenje

Kako bi se otkrio problem sa servisom, potrebno je vršiti njihovo stalno nadgledanje. Obrazac otkucaja srca se sastoji iz dva dela. Prvi deo obrasca potrebno je implementirati na samom servisu koji se nadgleda i koji služi za slanje poruka o stanju servisa.



Slika 10: Obrazac otkucaja srca

Svaki od servisa šalje poruke monitoru, u unapred definisanim vremenskim intervalima, *t1*.

Drugi deo obrasca je sam Monitor servis. On čuva sve otkucaje, odnosno poruke koje ostali servisi šalju. U njemu je definisano i period *t2* kao period na koje se čeka da bi se neka instanca javila. Komunikaciju izmedju servisa i monitora možemo videti na slici 10. Ukoliko neki od servisa nije poslao poruke duže od definisanog perioda, odnosno ukoliko ih Monitor nije primio, Monitor pokušava da komunicira sa servisom sinhrono, na primer preko HTTP-a. Ukoliko servis odgovori, znači da je sve u redu i potrebno je nastaviti izvršavanje. Ukoliko ga nema, onda se briše otkucaj, kako se više ne bi proveravao, i obavesti se supervizor servis kako bi razrešio novonastalu situaciju.

Trajanje *t2* bi trebalo uvek biti veće od *t1*, kako bi se dala prilika servisima da se javljaju, a ne da se stalno proverava od strane Monitora. Najbolje je ukoliko je ono par puta manje, tako da Monitor čeka da propusti nekoliko otkucaja od strane servisa kako bi pokrenuo svoj mehanizam za proveru ispravnosti servisa. Primer implementacije monitora možemo videti na primeru 7.

```
private class Monitor implements Runnable {
    private static final int VREME_CEKANJA = 15;

    @Override
    public void run() {
      proveri();
    }

    void proveri() {
      //dohvata sve istekle otkucaje odnosno one koji su stariji od
      //VREME_OD_POSLEDNJEG_OTKUCAJA_MINUTI
                                    List<Otkucaj>      istekli      =
dohvatiIstekleOtkucaje(VREME_CEKANJA);

      // proveri jos jednom da li je istekao otkucaj
      for (Otkucaj otkucaj : istekli) {
        String ipAdresa = otkucaj.dohvatiIPAdresu();
```

```
      // proveri da li je ziva
      boolean ziva;
      if (ipAdresa == null) {
        ziva = false;
      } else {
        //salje HTTP zahtev da proveri da li se servis javlja
        ziva = servisKlinet.daLiJeZiv(ipAdresa);
      }

      if (!alive) {
        //instanca ne odgovara, sklonimo otkucaj i pokrenemo
        //proceduru za oporavak ili obavestenje
        skloniOtkucaj(otkucaj);
        obavestiDaNedostaje(otkucaj.dohvatiServis());
      } else {
        //instanca je odgovorila na HTTP, i postavljamo trenutno
        //vreme kao novi otkucaj
        sacuvajVremeOtkucaja(otkucaj, now());
      }
    }
  }
```

Primer 7: Monitor u obrascu otkucaja srca

Posledice

Ovakav sistem daje dvostruku sigurnost da servisi rade. Prvi nivo provere je taj što servisi šalju svoje poruke Monitoru, kako bi on znao da su servisi živi i da rade kako treba. Ukoliko tih poruka nema, zbog neke mrežne greške ili servis preko kojeg se šalju nije dostupan, onda je na Monitoru kreiran drugi nivo provere. Monitor će u tom slučaju pokrenuti svoj mehanizam za proveru statusa. Ova provera bi morala biti sinhrona, dok poruke koje šalje servis mogu biti i asinhrone.

Unapeđivanjem samog sadržaja poruke koja se šalje može se unaprediti i ceo obrazac. Naime, sama poruka može biti modifikovana tako da sadrži podatke na osnovu kojih bi Monitor odlučivao kako radi servis i da li je potrebna spoljna intervencija na servisu. Na

primer, poruke koje sistem šalje mogu biti broj aktivnih zadatak koje servis obavlja, zauzetost procesora i memorije i slično. Tada bi kôd na servisu izgledao kao na primeru 8.

```
private class Otkucaj implements Runnable {

  @Override
  public void run() {
    posaljiOtkucaj();
  }

  void posaljiOtkucaj() {
    Otkucaj otkucaj = new Otkucaj();
    //broj taskova koji se izvrsavaju na instanci
    otkucaj.postaviBrojTaskova(dohvatiBrojTaskova())
    //zauzetost CPU
    otkucaj.postaviZauzetostCPU(dohvatiZauzetostCPU());
    //zauzetost Memorije
    otkucaj.postaviZauzetostMemorije(dohvatiZauzetostMemorije());

    red.posalji(otkucaj);
  }
}
```

Primer 8: Unapređen sadržaj poruke koja se šalje kao otkucaj

Takođe, ono što servis šalje kao zauzetost može se češće sakupljati i zatim slati, i na taj način stvoriti preciznija statistika šta se dešava na instanci. Monitor na osnovu ovih dobijenih poruka ne samo da može da oceni da li je instanca živa već i da li prima zadatke na izvršavanje, da li su procesi koji izvršavaju te zadatke živi i da li rade. Tim informacijama moguće je preciznije ustanoviti da li servis radi kako treba.

Dodavanje slanja otkucaja na strani servisa zahteva određene resurse i samim tim procesorska moć servisa može biti smanjena. Kada se implementira monitor, treba imati u vidu da broj poruka može biti ogroman, zavisno od intervala u kojem se šalju poruke i broja servisa koje on nadgleda. Ako monitor sporo obrađuje poruke, one se gomilaju i zapravo

monitor nikada nema tačno stanje o tome koji su sve servisi poslali poruke. Ovakvi problemi se često mogu izbeći korišćenjem obrasca obrade u serijama.

Jedna od većih pitanja u ovom obrascu jeste koje vreme uzeti kao vreme kada se neki servis javlja. Odnosno ko treba da odredi koje je to vreme, da li servis koji šalje otkucaje ili monitor.

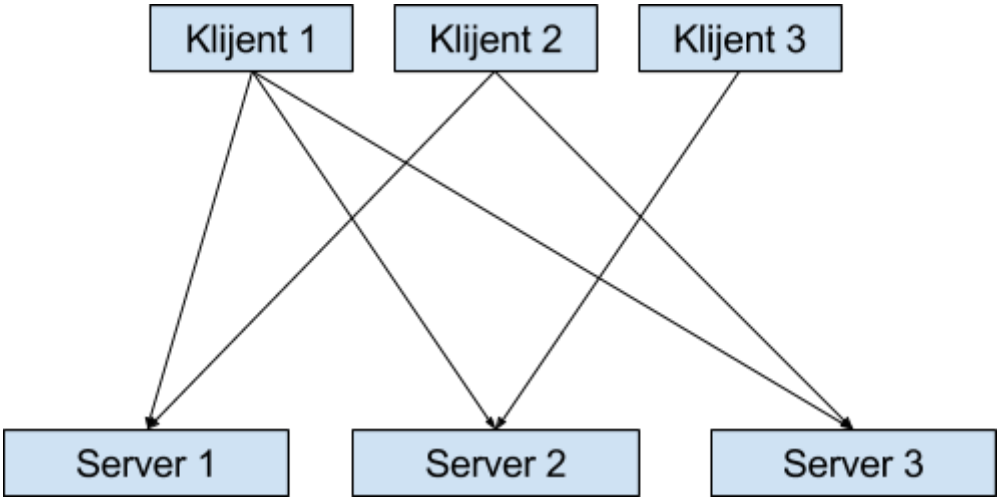
Prvi način je da sam servis šalje vreme zajedno sa porukom o otkucaju. Problem je što se serversko vreme može razlikovati od onog na monitoru, pa tako monitor nema tačne informacije prilikom određivanja da li je neki servis operativan ili ne. Drugi način je da sam monitor smatra da je vreme kada se servis javio zapravo trenutak kada on obrađuje to u poruku. U tom slučaju, usled nagomilavanja poruka, može se desiti da se kasno detektuje da neki servis ne šalje otkucaje. U svakom slučaju to je kompromis koji se mora napraviti i odlučiti se za jedno od dva rešenja.

2.2.8 Obrazac raspoređivača opterećenja

Problem

Aplikacije, iako pravljene da podržavaju veliki saobraćaj, vrlo često ne mogu da opsluže sve zahteve. Ograničenja mogu ići od problema u kodu, koji nije napisan za toliki broj zahteva, do samih fizičkih opterećenja mreže ili servera.

Način na koji je moguće rešiti ovaj problem je ukoliko se broj servisa, odnosno servera, uveća. Tako prostim umnožavanjem dobijamo veći kapacitet sistema, i to za onoliko servera koliko smo ih dodali. Međutim problem kod ovakvog rešenja je što postoji više tačaka koje korisnik može da pita za obradu zahteva. Ukoliko postoje tri servera koji opslužuju zahteve, kao na slici 11, svaki od klijenata može poslati zahtev bilo kom od njih. Ukoliko svi šalju istom serveru, druga dva ostaju prazna i tako ostaje ista situacija kao i da nismo povećavali broj servera.

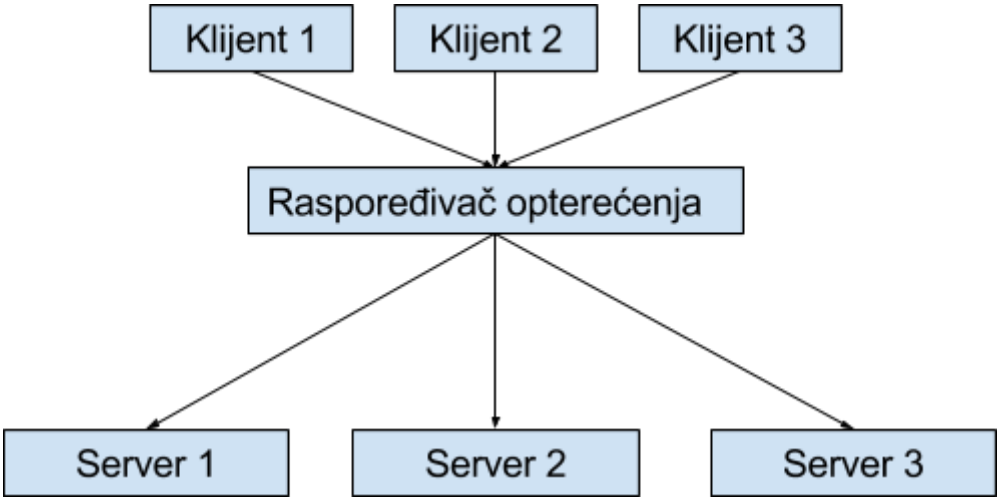


Slika 11: Arhitektura sa više servera

Na ovaj način se ne postiže da serveri budu optimalno i ravnomerno opterećeni, već je isključivo na klijentima koji će server koristiti. To može dovesti da jedan od servera radi na granicima svojih kapaciteta, dok drugi može biti sasvim neiskorišćen.

Rešenje

Ukoliko se uvede dodatna komponenta, koja će primati sve zahteve od klijenata i onda ih prosleđivati jednom od servera, dobijamo servis koji se zove raspoređivač opterećenja (silka 12) i koji je osnovni princip ovog obrasca.



Slika 12: Obrazac raspoređivanja opterećenja

Raspoređivač opterećenja možemo implementirati na nekoliko načina, ali svi oni imaju jedan zajednički princip funkcionisanja:

Primi zahtev - odredi kojem serveru proslediti zahtev - prosledi zahtev.

Ovaj kôd moguće je implementirati na više nivoa, od preusmeravanja samog TCP saobraćaja, do primanja HTTP zahteva, prosleđivanja i vraćanja odgovora. Međutim, ono što ih zapravo razlikuje, sa funkcionalne a ne implementacione strane, je kriterijum po kom biraju kojem serveru proslediti zahtev.

Osnovni i najjednostavniji raspoređivač je svakako *Round-robin*. Odnosno raspoređivač koji će imati niz svih raspoloživih servisa i onda dodeljivati zahteve prvom, zatim drugom, trećem i nakon toga ponovo, iz početka. Primer 9 sadrži takvu implementaciju.

```
public class Rasporedjivac {
  //lista svih raspolozivih servisa
  private List<Servis> servisLista = new ArrayList<Servis>(
      Arrays.asList(Servis1, Servis2, Servis3));
  //servis koji ce obraditi sledecu poruku
  private int sledeciServis = 0;

  public void obradiZahtevRoundRobin(Zahtev zahtev){
    //dohvati servis koji treba da obradi zahtev
    Servis servis  = servisLista[sledeciServis];
    //obradi zahtev
    servis.prosledi Zahtev(zahtev);
    //povecaj brojac kako bi sledeci servis uzeo novi
    sledeciServis++;
  }
}
```

Primer 9: Raspoređivač sa *Roundrobin* principom

Na ovaj način svi servisi dobijaju jednak broj zahteva. Rezultat ovakve strategije je sličan kao i da je izbora servisa bio nasumičan, ukoliko je raspodela funkcije izbora servisa uniformna.

Ipak, problem kod ovakve postavke pojavljuje se kada nisu svi zahtevi iste težine, odnosno kada trajanje izvršavanja nije jednako. Onda je moguće da se na određenom servisu nagomila više poruka čija obrada traje dugo. Nagomilavanjem takvih zahteva jedan server postaje preopterećen, dok drugi koji je primao zahteve koji traju relativno kratko ostaje neiskorišćen. Da bi se izbeglo takvo ponašanje neophodno je implementirati bolji način raspoređivanja. Jedan od načina je da se pored spiska servera, u raspoređivaču čuva i koliko je koji od servisa opterećen. Ovo se može uraditi na sličan način kao što smo opisali u obrascu otkucaja srca. Odnosno tako što bi svaki server slao podatke o opterećenosti raspoređivaču. Raspoređivač na osnovu tih podataka uvek može da izabere server koji je najmanje opterećen u tom trenutku i istom prosledi zahteve.

```
public class Rasporedjivac {

  public void obradiZahtevRoundRobin(Zahtev zahtev){
    //dohvati servis koji treba da obradi zahtev
    Servis servis  = dohvatiNajmanjeOpterecenServis();
    //obradi zahtev
    servis.prosledi Zahtev(zahtev);
  }

    public void primiOpterecenje(ServisId servisId, Opterecenje
opterecenje){
    osveziOpterecenjeServisa(servisId, opterecenje);
  }
}
```

Primer 10: Raspoređivač sa faktorom opterećivanja servisa

Na primeru 10 je prikazan takav slučaj, u kojem se na taj način svaki server opterećuje optimalno, koliko je to moguće. Uvek se najmanje opterećen servis favorizuje za nove zadatke, i tako nijedan servis ne radi na maksimalnom opterećenju dok ne dođe do opterećenja celokupnog sistema.

Ukoliko znamo koliko je zahtevan posao koji dobijamo, tada je moguće bolje odrediti na koji server ga proslediti. Ovo naročito dolazi do izražaja u okruženju u oblaku, jer je tamo vrlo lako, i verovatno, postojanje različitih tipova servera. Naime, Google nudi 21 različit tip instanci, Microsoft Azure 48, dok Amazon nudi čak 76 tipova. Tako je moguće podesiti sistem, da se optimalno koriste resursi mašine..

Kod takvih slučajeva u raspoređivaču je potrebno implementirati deo, koji analizira sam sadržaj poruke, kao na primeru 11.

```
public class Rasporedjivac {

  public void obradiZahtevRoundRobin(Zahtev zahtev){
    //dohvati resurse koji su neophodni za zahtev
    Resursi resursi = dohvatiResurse(zahtev);
    //dohvati servis koji treba da obradi zahtev na osnovu resurs
    Servis servis  = dohvatiServisSaResursima(resursi);
    //obradi zahtev
    servis.prosledi Zahtev(zahtev);
  }
}
```

Primer 11: ispitivanje sadržaja poruke kod raspoređivača

Posledice

Kod metode `dohavtiResurse(Zahtev zahtev)` važno je da ona mora imati jako dobre performanse, odnonos veliku brzinu izvršavanja. Naime, raspoređivač nikako se ne sme opteretiti teškom i zahtevnom logikom prilikom raspoređivanja, jer u tom slučaju on može postati usko grlo, i postati nedostupan. Prilikom implementacije moramo imati u vidu da kapacitet cele platforme može biti ugrožen, odnosno kapacitet je ograničen kapacitetom samog raspoređivača.

S obziorom da je raspoređivač centralni deo sisitema i da svi zahtevi prolaze kroz njega tu leži i najveća opasnost odnosno mana ovog obrasca. Kao što je već naglašeno kapacitet celog sistema zavisi od njega, ali i performanse celog sistema takođe zavise od tog servisa. Međutim, najveća mana je pre svega to što je on takozvana jedinstvena tačka pucanja. To znači da ukoliko ovaj servis ne radi neće raditi ceo sistem.

Iz tog razloga inženjeri uvek biraju proverene sisteme, sa dobrim performansama, kada razmišljaju o uvođenju raspoređivanja opterećenja. To su prepoznali i sami poslužioci računarstva u oblaku i ponudili svoja rešenja. Sada je moguće koristiti ovaj obrazac u svega

par koraka, jednostavnim konfigurisanjem servisa. Ovo govori o tome koliko je ovaj obrazac bitan i prisutan u oblaku.

Amazon nudi servis koji nazivaju *Elastic Load Balancer* (ELB). ELB automatski distribuira dolazeći saobraćaj na više instanci. Omogućava da se postigne puna otpornost na greške, lako obezbeđujući nepohodni kapacitet sistema za distribuciju opterećenja kako bi se obrađivao sav saobraćaj.

Microsoft Azure takođe ima servis implementiran, i naziva se *Azure Load Balancer* (ALB). On nudi visoku dostupnosti i performanse mreze. Implementiran je na četvrtom nivou mrežnog protokola (TCP, UDP) koji distribuira saobraćaj ka zdravim instancama.

Slično rešenje nudi i *Google*, koji se naziva *Google Cloud Load Balancing* (GCLB). Sa GCLB-om, jedna IP adresa opslužuje sve servise na *backend*-u. Obezbeđuje distribuciju ne samo između instanci nego i između regiona, tako da ukoliko dođe do problema u jednom regionu ceo saobraćaj je moguće preusemriti na drugi region. GCLB za razliku od drugih nudi i distribuciju na osnovu sadržaja, koja može biti korisna kao što smo i pisali o tome.

2.2.9 Obrazac automatskog skaliranja

Problem

Obrazac raspoređivanja opterećenja je koristan kada je potrebno da se servisi ravnomerno opterete. Međutim, kod takvih sistema uvek je neophodan dovoljan broj servera koji rade u istom trenutku kako bi se obezbedilo dovoljno resursa. Sistem mora imati dovoljan kapacitet i prilikom najvećeg saobraćaja, da bi se opslužili svi korisnici i vratili rezultati. Naravno, jako je skupo imati uvek dovoljno servera koji stalno rade da bi opslužili i najveći saobraćaj. Ti serveri rade punim kapacitetom samo par puta u danu, nedelji ili mesecu, zavisno od sistema, ali ih je potrebno održavati i plaćati sve vreme, iako bi bilo sasvim dovoljno da radi samo neki od njih u većini slučajeva.

Osnova poslovanja svakog poslužioca računarstva u oblaku pre svega je da nudi svoju infrastrukturu ili servise na određeno vreme. Svi oni nude zakup različitih servera, pozivajući njihve servise dobija se instanca na kojoj je lako postaviti sve željene servise i aplikacije. Istu instancu moguće je dobiti u bilo koje vreme, koristiti koliko je neophodno i kada više nije potreban vratiti je nazad poslužiocu.

Amazon nudi sedeamdeset i šest tipova instanci, od onih sa jednim jezgrom i 0.5GB memorije, do onih sa 128 jezgara i skoro 2TB memorije (1952GB). Iste te instance moguće je "iznajmiti" na par sati, do više dana, meseci, godina, i platiti ih u skladu sa tim. Na taj način u svakom trenutku moguće je izabarati instancu koja odgovara datom problemu u datom trenutku. *Microsoft Azure* zapravo nudi sličan proizvod *Amazon*-ovom, sa razlikom u tipu i jačini samih servera.

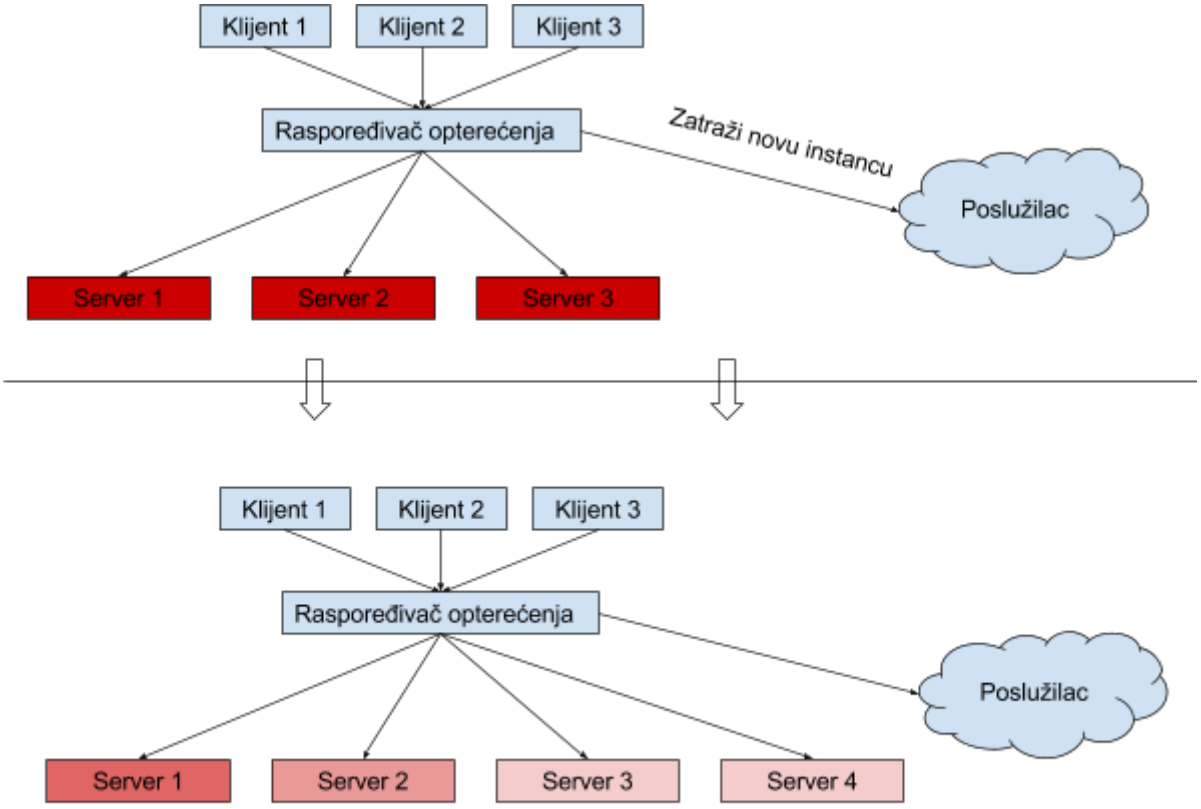
Google takođe nudi dinamičko podizanje instanci, izborom nekog od dvadeset i jednog tipa. Razlika, pored same veličine i broja tipova instanci, jeste i u tome što *Google* nudi naplatu po minutu, minimalno 10 minuta. Na taj način kreiranje sistema može biti još fleksibilnije, i zaista koristit samo neophodne resurse.

Rešenje

Ukoliko imamo nekoliko servera koji opslužuju zahteve, prilikom velikog opterećenja možemo dobiti dodatni prostor za skaliranje tako što bi na postojeći sistem dodali novi server koji dinamički podižemo od strane oblaka.

Naravno ovaj sistem nema punu snagu i svrhu ukoliko se implementira bez obrasca raspoređivanja opterećenja. Samo dodavanje servera ne pravi veliku razliku ukoliko nema raspoređivača koji bi prosledio dodatan posao na taj server. Najbolji rezultati dobijaju se ukoliko je raspoređivač implementiran sa metodom prosleđivanja serveru koji je najmanje opterećen. Na taj način, novo dodati server će primati sav saobraćaj dok se ne približi ostalim serverima po zauzetosti. Ukoliko postoje tri servera koji rade na granicama mogućnosti, kao na slici 13, dodavanjem novog servera može se smanjiti opterećenost trenutnih servera i podići kapacitet celokupnog sistema.

Ovaj sistem dobija svoju punu snagu kada se spoji sa obrascima raspoređivanja opterećenja i otkucaja srca. Odnosno, raspoređivač opterećenja bi trebalo da na neki način donese odluku kada da doda novu instancu. Taj događaj može biti pokrenut intervencijom administratora sistema, ali jasno je da takav sistem ne može da da optimalne rezultate. Ukoliko implementiramo obrazac otkucaja srca, gde svaki od servera šalje svoje stanje, ali i koliko je zauzet, onda servis može doneti informisanu odluku kada podići novi servis. U slučaju da raspoloživi servisi prevaziđu prag opterećenosti koji je definisan, potrebno je pokrenuti proceduru dodavanja novog servisa, kao u primeru 12.



Slika 13: Obrazac automatsko skaliranja

```
private class Skaliranje implements Runnable {
    private static float MIN_OPTERECENJE_POKRETANJA = 0.7;
    private static float MAX_OPTERECENJE_POKRETANJA = 0.95;
    private Map<String,Object> instanceNaRasporedjivacu =
                                                    new HashMap<>;
    @Override
    public void run() {
      skaliraj();
    }

    private void skaliraj() {
      while(true){
        Instanca instanca;
```

```
        //dohvati sve poslednje otkucaje servisa
        ArrayList<Otkucaj> otkucaji = dohvatiSveOtkucaje();
        Collections.sort(otkucaji);
        //ukoliko je opterecenost najmanje opterecene instance
        //veci od MIN_OPTERECENJE_POKRETANJA
        //i najopterecenije vece od MAX_OPTERECENJE_POKRETANJA
        //podigni novu instancu
        if (otkucaji.get(0).getOpterecenje() >
MIN_OPTERECENJE_POKRETANJA && otkucaji.get(otkucaji.size()-1) >
                                        MAX_OPTERECENJE_POKRETANJA)
{
          try {
            instanca = pokreniNovuInstancu();
            inicijalizujInstancu(instanca);
            nakaciInstancuNaRasporedjivac(instanca);
            instanceNaRasporedjivacu.put(instanca.dohvatiId(),
instanca);
          } catch (Exception e) {
            //zapisi ukoliko nije moguce podici instancu i nastavi
            System.out.println("UPOZORENJE: nije moguce dobiti
                                                    instancu" +
e);
            //iskljuci instancu ako smo je dobili
            if(instanca != null){
              iskljuciInstancu();
            }
          }
        }
      //sacekaj sekund pre nego sto ponovo proveri da li je
      //potrebno skalirati
        Thread.sleep(1000);
      }
    }
}
```

Primer 12: Dodavanje novih instanci u automatskom skaliranju

Odluku o dodavanju novog servisa ne treba doneti na osnovu zauzetosti samo jednog servera. Server koji se meri, može imati povećanu zauzetost zbog neke posebne situacije, ali i tada ostali serveri ipak mogu da izdrže dodatne zahteve. Zato je najbolje implementirati granicu koju moraju preći najviše zauzet server i onaj sa najmanjom zauzetošću. Tek ukoliko su oba kriterijuma zadovoljena treba pokrenuti novi server i dodati ga u raspoređivač. Naravno, treba imati u vidu da se procedura ne pokrene prekasno, kada je opterećenost sistema 100%, jer je serverima potrebno vreme da se pokrenu i krenu da preuzimaju zadatke.

Takođe, ne mora se dodavati po jedna instanca istovremeno. Potrebno je definisati koliko će se instanci podizati u kojim slučajevima. Na primer, ukoliko je zauzetost preko 80% podići jedan novi server, a ukoliko je zauzetost preko 90% procenata onda podići dva nova servera.

Povećanje broja servera pomaže da se izdrži velika potražnja za izvršavanjem zahteva, međutim da bi se smanjila cena potrebno je dodati i deo koji izbacuje instance i gasi ih ukoliko je sistem slabo iskorišćen, odnosno nakon što dođe do pada saobraćaja. Za razliku od povećanja, kada je dovoljno da primetiti rast, kod pada saobraćaja to nije dovoljno. Naime, često se dešava da saobraćaj dolazi u naletima, i to u kratkim vremenskim intervalima. Kada bi se povećavao i smanjivo broj servera prilikom svake promene, došlo bi do velikih troškova. Zato je najprimerenije imati strategiju takvu da ukoliko se primeti da je na velikom broju servera zauzetost manja od nekog praga duže od *n*, tada skloniti taj server iz raspoređivača i ugasiti instancu.

Oduzimanje, odnosno izbacivanje jednog servera sa sobom nosi još jednu specifičnost. Naime, ukoliko se na tom serveru već dešavaju neki procesi, odnosno ako se neki zahtevi već obrađuju, ne bi bilo dobro da se server ugasi jer bi zahtevi pali, odnosno ne bi bili izvršeni. Potrebno je, da kada se odluči da se neka instanca izbaci, prvo je odseći, odnosno ne puštati nove zahteve ka njoj. Tek kada je instanca prazna treba je ugasiti.

```
private class Skaliranje implements Runnable {
  private static float MIN_OPTERECENJE_POKRETANJA = 0.7;
  private static float MAX_OPTERECENJE_POKRETANJA = 0.95;
  private static float PROCENAT_SERVERA_ZA_PROVERU = 0.5;
  private static float PRAG_ZA_SMANJENJE = 0.3;
  private Map<String,Object> instanceNaRasporedjivacu =
                                          new HashMap<>;
    private  Map<String,  Object>  instanceZaZaustavljanje  =  new
```

```
HashMap<>;
  @Override
  public void run() {
    skaliraj();
    smanjiBrojInstanci();
  }


  private void skaliraj() {...}

  private void smanjiBrojInstanci() {
    while(true){
      //dohvati sve poslednje otkucaje servisa
      ArrayList<Otkucaj> otkucaji = dohvatiSveOtkucaje();
      Collections.sort(otkucaji);
      //proveri prosecnu zauzetost PROCENAT_SERVERA_ZA_PROVERU
      //najmanje zauzetih servera
      //koje ispitujemo daa li je pala zauzetost
      float zauzetost = izracunajProsecnuZauzetost(otkucaji,

PROCENAT_SERVERA_ZA_PROVERU);

      //ako je zauzetost manja od praga
      if(zauzetost < PRAG_ZA_SMANJENJE){
        //skloni instancu da ne dobija nove taskove
        instanceNaRasporedjivacu.remove(instanca.uzmiId());
        //stavi je na red za zaustavljanje
        instanceZaZaustavljanje.put(instanca.uzmiId(), instanca);
      }

      dealocirajSpremne();
      //sacekaj minut pre nego sto ponovo proveri da li je potrebno
      //skalirati
      Thread.sleep(60000);
    }
```

```
  }


  private void dealocirajSpremne(){
    for(Instanca instanca : instanceZaZaustavljanje){
      if(instanca.prazna()){
        dealociraj(instanca);
        instanceZaZaustavljanje.remove(instanca.uzmiId())
      }
    }
  }
}
```

Primer 13: Izbacivane instanci iz upotrebe u sistemu sa automatski skaliranjem

S obzirom da se instance čuvaju i koriste dokle god imaju posla, odnosno dokle god se na njima izvršavaju poslati zahtevi, možemo ih iskoristiti za potencijalno pokretanje novog servera. Odnosno ukoliko se ustanovi da je neophodna nova instanca, umesto zahteva poslužiocu oblaka prvo je potrebno proveriti da li postoji instanca u mapi instanci za gašenje, i iskoristiti je ukoliko postoji. Tada metoda `alocirajNovuInstancu()` u metodi `skaliraj()` izgleda:

```
private Instanca alocirajNovuInstancu() throws Exception{
    Instanca instanca;
    //ukoliko postoji instanca koja je kandidat za zaustavljanje
    if (!instanceZaZaustavljanje.isEmpty()){
                                                    instanca        =
instanceZaZaustavljanje.entrySet().get[0].getValue();
      //skloni instancu sa spiska za zaustavljanje
      instanceZaZaustavljanje.remove(instanca.uzmiId());
    } else {
      //ukoliko ne postoji, alociramo novu od provajdera
```

```
      try{
        Instanca instanca = podigniInstancu();
      }
    }
    //dodaj instancu za rasporedjivanj
    instanceNaRasporedjivacu.put(instanca.uzmiId(), instanca);
    return instanca;
  }
```

Primer 14: Alociranje sa skupom raspoloživih instanci

```
private void ugasiSpremne(){
//dohvati sve instance za gasenje
for(Instanca instanca : instanceZaGasenje){
  //za svaku proveri da li je moguce gasiti
  if(daLiJeSpremnaZaGasenje(instanca)){
    ugasi(instanca);
    instanceZaGasenje.remove(instanca.uzmiId())
   }
}

private boolean daLiJeSpremnaZaGasenje(Instanca instanca) {
  //ukoliko je instanca prazna
  //i ukoliko je vreme trajanja po modulu 60 vece od 50
  //instanca je spreman za gasenje
  if(instanca.prazna() &&
       (instanca.dohvatiVremeTrajanjaMin() % 60) > 50){
  return true;
}
return false;
}
```

Primer 15: Gašenje do punog intervala

Na taj način zapravo pravimo skup instanci u kome uvek prvo potržimo novu instancu, pa tek ako ne postoje zahtev prosleđujemo poslužiocu oblaka. Ovo je moguće dodatno unaprediti, naročito kod onih provajdera kod kojih se period naplate zaokrožuje do punog sata. Takvi su *Amazon* i *Azure*, za razliku od Google-a koji naplaćuje po minutu. Dodatno unapređenje se svodi na to da se instanca koja je kandidat sa sklanjanje, gasi samo ako je blizu punog sata. Odnosno ukoliko je instaca korišćena 15 minuta, sačekati još 40 min ako se stvori potreba za novom, a tada vratiti tu instancu na korišćenje. Tada bi kod metode `ugasiSpremne()` bio izmenjen na način kao u primeru 15.

Automatsko skaliranje donosi velike uštede jer se serveri koriste samo kada je to potrebno. S obzirom na svoju isplativost jedan je od najkorišćenijih obrazaca u oblaku, što je uviđeno i od strane poslužilaca računarstva u oblaku.

Amazon u okviru svog ELB nudi opciju automatskog skaliranja. Potrebno je definisati tip servera koji se podiže u slučaju povećanog saobraćaja. Takođe, prilikom kreiranja ELB-a navode se i pravila za skaliranje na gore i dole, odnosno kada da *Amazon* podigne nove instance, a kada da skloni postojeće.

Azure nudi sličan sistem, nudeći skaliranje koje može biti izvršeno na osnovu predefinisanog rasporeda ili neke metrike poput zauzetosti procesora ili memorije. Na primer, takva pravila mogu biti:

- Podići 10 instanci radnim danima i smanji na 4 vikendom
- Podiči po jednu instancu ukoliko je prosečna iskorišćenost procesora preko 70%, a gasiti ukoliko zauzetost procesora padne ispod 50%
- Podiči novi server ukoliko je broj poruka na nekom serveru ili redu preko određenog praga

Kada se podižu nove instance, potrebno je vreme kako bi se dobile na korišćenje, instalirali operativni sistem, *driver*-i i ostali softveri nepohodni za rad. Ovaj proces može da traje od minut do 10 minuta, zavisno od zahteva, potražnje a i samog poslužioca. Kod automatskog skaliranja to može izazvati dosta problema, jer je neophodno predvideti povećani saobraćaj 10 minuta ranije nego što on prevaziđe raspoložive kapacitete. Ono što *Amazon* nudi je, takozvano, zagrevanje ELB, odnosno instanci potrebnih za rad ELB-a. Naravno, tako nešto je neophodno najaviti *Amazon*-u, i nije rešenje na duge staze, već samo kada se unapred zna da će u otačno dređenom periodu postojati povećan saobraćaj. Sa „zagrejanim" instancama, proces dodavanja nove instance skoro da je trenutan.

Google-ovo rešenje se razlikuje upravo po tome što skalira u realnom vremenu. Odnosno kako raste broj zahteva, pa čak iako neočekivno naiđe veliki broj, *Google* automatsko skaliranje ne zahteva zagrevanje već se skaliranje od nule do punog kapaciteta dešava u par sekundi.

Posledice

Većina sistema nema stalnu opterećenost. Saobraćaj i broj zahteva dolazi u talasima. Planiranje kada i koliko resursa je neophodno za podršku celokupnog saobraćaja zahteva puno podataka i analize. Obrazac automatskog skaliranja eliminiše potrebu za detaljnim planiranjem jer se u njemu daje mogućnost da sistem sam u zavisnosti od trenutne potražnje, poveća ili smanji dostupne resurse. Ne samo da je planiranje i održavanje takvog sistema jednostavnije, već su i troškovi takvog sistema manji jer u svakom trenutku je na raspolaganju onoliko resursa koliko je neophodno, ni manje ni više.

Kada se implementira ovakav sistem, mora se imati u vidu da zahtevi koji se dugo izvršavaju mogu biti potencijalni problem. Zbog takvih zahteva nije moguće jednostavno izbaciti server iz upotrebe, jer bi zahtevi prestali da rade. Zato je potrebno ili prihvatiti činjenicu da neki zahtevi mogu prestati u bilo kom trenutku, ili da se instanca nikada ne gasi dokle god na njoj ima zadataka koji se izvršavaju. Međutim, to može uvećati troškove. Najoptimalniji, ali ujedno i najkompleksniji, način za rešavanje ovakve situacije je da se aplikacija, ukoliko je to moguće, implementira tako da je moguće preseliti zadatke sa jednog na drugi server, tokom izvršavanja. Nakon što se prebace svi zahtevi sa jednog servera, onda se bez problema server može isključiti.

Glavna mana automatskog skaliranja je to što se sistem izlaže potencijalno, neželjno velikom trošku. Ukoliko saobraćaj stalno raste, ili je jako velik, sistem će stalno dodavati nove instance, stvarajući velike troškove. Ovo posebno može doći do izražaja u slučaju zlonamernog napada na sistem. Napadač jednostavnim generisanjem velikog saobraćaja podstiče sistem da podigne nove servere. Da bi se ovo sprečilo moguće je ograničavanje maksimalnog broja servera, što znači da sistem ipak ima ograničenje u skaliranju. Svakako, uvođenje mera zaštita od neželjenih napada, poput DDoS napada, je apsolutno neophodno kod ovakvih sistema.

2.2.10 Obrazac prikolice

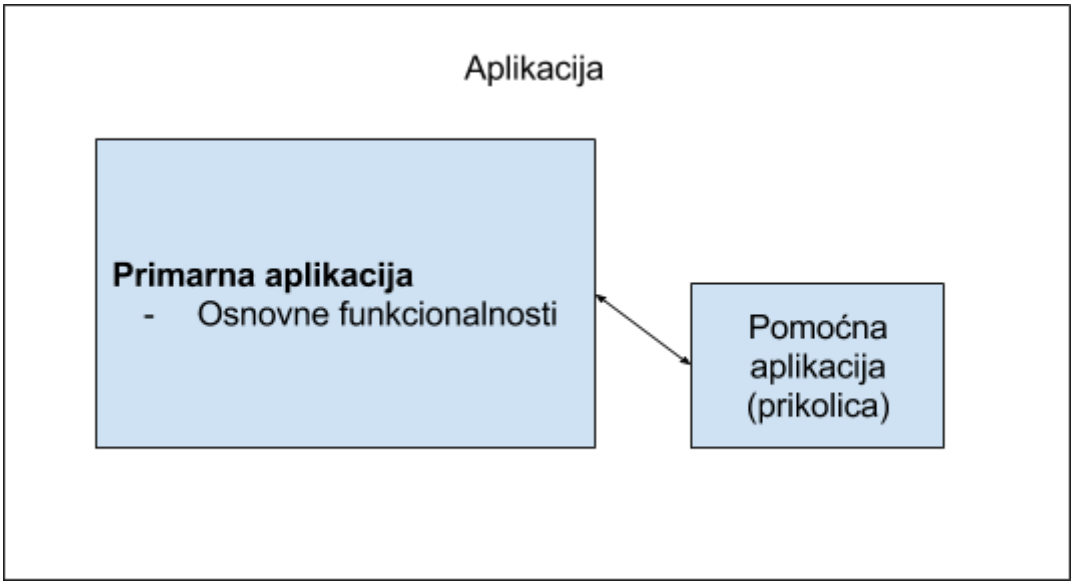
Problem

Servisi i aplikacije vrlo često zahtevaju dodatan skup funkcionalnosti koje idu uz osnovnu aplikaciju. Takve funkcionalnosti mogu biti ispisivanje i sakupljanje dnevnika izvršavanja (logova), nadgledanje rada servisa, provera ispravnosti, mrežni servisi i slično. Ove aplikacije ne utiču na samo izvršavanje zadataka koje glavni servis obavlja ali ga dopunjuju i nude niz servisa i izveštaja koje olakšavaju razvoj i nadgledanje samog servisa. Pomoćni servis deli životni ciklus sa glavnim servisom, i pruža dodatni skup funkcionalnosti kako se ne bi opterećivao glavni servis. Kako pomoćni servis zapravo proširuje skup funkcionalnosti, ali obavezno ide zajedno sa glavnim servisom, otuda i naziv obrazac prikolice. Izdvajanjem ova dva servisa podržava se izolacija i enkapsulacija. Način rada ovih aplikacija je moguć ili podizanjem više procesa ili zasebnih kontejnera u kojima se oni izvršavaju.

Rešenje

Ukoliko servis zahteva niz funkcionalnosti koje nisu ključne za samo izvršavanje zadataka koje ima, kao što su logovanje, konfigurisanje i slično, ti servisi se mogu izmestiti u poseban servis, kao što je ilustrovano na slici 14. Taj poseban servis je moguće razviti posebno i pustiti u rad, iako je on usko vezan za funkcionalnost osnovne aplikacije. Rad takva dva servisa se naziva obrazac prikolice.

Sve dodatne funkcionalnosti koje pruža pomoćni servis moguće je implementirati i u osnovnom servisu, međutim kako bi oni delili iste resurse, može doći do gubitka u performansama osnovnog servisa.



Slika 14: Obrazac prikolice

Ovaj obrazac zapravo i jeste podobrazac mikroservis obrasca. On takodje zagovara razdvajanje većeg, robusnijeg sistema na više manjih, jednostavnijih servisa. Međutim, u obrascu prikolice servisi nisu sasvim nezavisni. Pomoćna aplikacija nema previše smisla, i ne pruža dovoljno funkcionalnosti, bez primarne aplikacije, osim ukoliko je njen posao da obavesti da primarna aplikacija ne funkcioniše. U većini slučajeva ove dve aplikacije zapravo dele i životni vek.

Ovaj obrazac je pogodan i u slučaju kada nešto ne funkcioniše kako bi trebalo u pomoćnom delu sistema, jer glavna aplikacija može da nastavi da radi čak iako je izgubljena neka od funkcionalnosti pomoćne aplikacije. Na primer, ukoliko je servis za skupljanje logova prestao da radi i logove nije moguće poslati, nema razloga da se glavni servis bavi oporavkom od tih grešaka i eventualno da rizikuje performanse ili sam integritet aplikacije. Ukoliko je moguće takvu operaciju uopšte obaviti, to će uraditi pomoćna aplikacija, dok osnovni servis nastavlja da funkcioniše normalno.

Posledice

Ukoliko se implementira obrazac prikolice, onda je moguće da se i sami servisi napišu na potpuno različite načine. Odnosno, svaki od servisa može biti napisan u onom jeziku koji najviše pogoduje problemu koji rešava, kao i da koristi tehnologije koje ne zavise od drugih funkcionalnosti celog sistema. S obzirom da se i same aplikacije mogu razlikovati po tehnologiji, pri razvijanju sistema treba voditi računa o protokolima za komunikaciju. Izmene

istih treba svesti na neophodni minimum, kako bi dalji razvoj i održavanje komponenti bilo što lakše i jednostavnije.

Ovaj obrazac nema smisla u aplikacijama koje su dosta male, i kod kojih bi razdvajanje samo uvelo dodatnu kompleksnost i otežalo dalje održavanje. Takodje, ukoliko ova dva pod servisa treba da skaliraju različitom brzinom ili obimom, onda će više smisla imati da se dva servisa potpuno odvoje.

3. Zaključak

Sistem u oblaku moguće je dizajnirati i i implementiran na više načina. Svakako najlakši, sa stanovišta vremena i resursa koje je potrebno uložiti u prelazak sa tradicionalnog sistema je njegovo preslikavanje. Svi tradicionalni obrasci koji su se koristili važe i sada, u oblaku, i daju slične rezultate koje su davali i pre.

Međutim, oblak nudi mnogo više. Rast broja alata koje on pruža, povećanje dinamičnosti i fleksibilnosti sistema sa sobom nose i odlične mogućnosti. Sistemi sada mogu iskoristit nove alate, kako bi dostigli nove nivoe skalabilnosti, fleksibilnosti i pouzdanosti. Ovi alati olakšavaju kreiranje sistema, ali kako se otvaraju nove mogućnosti i novi zahtevi bivaju sve ambiciozniji.

Razmišljanje o sistemima u oblaku najbolje je prilagoditi novom, dinamičnijem okruženju. Razmišljanje kroz prizmu obrazca mikroservisa otvara velike mogućnosti ka kreiranju novih rešenja. Povećanje broja servisa diktira da se i sama komunikacija menja, pa tako i način na koji se obrađuju svi ti događaji mora se prilagoditi. Kroz obrasce asinhronih redova i obrade u serijama dobija se dosta na rezultatima samog sistema sa promenom načina kako se poruke obrađuju. Kako servisi trpe sve veće opterećenje, pristup rešavanju problema sistemom „podeli pa vladaj" daje dobre rezultate. Tako u sprezi obrazaca mikroservisa sa raspoređivačem opterećenja i automatskog skaliranja i respoređivača opterećenja sistem postaje fleksibilniji.

Svi obrasci pored očiglednosti koje nose sa sobom, nose i cenu koja se mora platiti. Većina opisanih servisa unosi dodatne korake u izvršavanjima. Taj dodatni korak zna uvesti i dodatna kašnjenja, tako se prilikom uvođenja istih mora razmišljati i o tome. Prilikom odlučivanja o implementaciji nekog obrasca, moraju se imati u vidu i njegove loše strane, ali i kako taj obrazac utiče na ostale delove sistema i kako će uticati na budući razvoj.

Sistemi u oblaku se razlikuju od tradicionalnih distribuiranih sistema. Po definiciji, distribuiran sistem je skup nezavisnih računara koji se predstavljaju korisniku sistema kao jedinstveni računar. Sa druge strane, računarstvo u oblaku se sastoji, a još važnije, prepoznaje postojanje mnogo manjih, nezavisnih i nepouzdanih komponenata koje mogu biti povremeno nedostupne.[20] Ta razlika uslovljava složenije obrasce za optimalnije funkcionisanje, ali i nove mogućnost.

Zato novi obrasci koji su prilagođeni oblaku zapravo daju pravu snagu oblaku i stvaraju novi način razmišljanja u računarstvu. Primenom novih obrazaca i snagom oblaka, sistemi postaju sve sposobniji za obavljanje i najkomplikovanijih zadataka, koji su u prošlosti delovali nemoguće i otvaraju put ka novim otkrićima.

Literatura

[1] Gordon E. Moor, **Cramming more components onto integrated circuits**, *Electronics*, Volume 38 (8), 114-117, 1965.

[2] Chip Walter, **Kryder's Law**, *Scientific American*, Volume 293, 32-33, 2005.

[3] Rich Tehrani, **As we may communicate**, TMCNet, http://www.tmcnet.com/articles/comsol/0100/0100pubout.htm, 20.07.2017.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia, **A View of cloud computing**, *Communications of the ACM*, Volume 53, 50-58, 2010.

[5] Hassab Quasay, **Demystifying Cloud Computing**, *The Journal of Defense Software Engineering*, Volume 11, 16-21, 2008

[6] Amazon, **Resizing Your Instance**, *Amazon AWS*, http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-resize.html, 25.08.2017.

[7] Google, **Changing the Machine TYpe of a Stopped Instance**, *Google Cloud*, https://cloud.google.com/compute/docs/instances/changing-machine-type-of-stopped-instance, 25.08.2017.

[8] Microsoft, **Resize virtual machines**, *Microsoft Azure*, https://azure.microsoft.com/en-us/blog/resize-virtual-machines/, 25.08.2017.

[9] Christopher Alexander, **A pattern Language: Towns, Buildings, Constructions**, *Oxford University Press, New York*, 1977., ISBN 978-0-19-501919-3

[10] Linda Rising, **The Patterns Handbook: Techniques, Strategies, and Applications**, *Cambridge University Press, Cambridge*, 1998., ISBN 0-521-64818-1

[11] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, Trent Swanson, **Cloud Design Patterns**, *Microsoft*, 2014., ISBN 978-1-62114-036-8

[12] Thomas Erl, Robert Cope, Amin Naserpour, **Cloud Computing Design Patterns**, *Prentice Hall*, 2015., ISBN: 978-0-13-385856-3

[13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, *Pearson Education*, 1994., ISBN: 978-0201633610

[14] Amazon, **Amazon Simple Queue Service**, *Amazon AWS*, https://aws.amazon.com/sqs/ , 27.07.2017.

[15] Microsoft, **Service Bus Messaging**, *Microsoft Azure*, https://docs.microsoft.com/en-us/azure/service-bus-messaging/, 27.07.2017.

[16] Google, **Cloud Pub/Sub**, Google Cloud, https://cloud.google.com/pubsub/, 27.07.2017.

[17] Amazon, **Amazon EC2 Pricing**, *Amazon AWS,* https://aws.amazon.com/ec2/pricing/on-demand/, 28.07.2017.

[18] Microsoft, **Azure Pricing**, *Microsoft Azure, https://azure.microsoft.com/en-us/pricing/, 29.08.2017.*

[19] Google, **Google Pricing**, *Google Cloud*, https://cloud.google.com/pricing/, 29.08.2017.

[20] Jinquan Dai, Bo Haung, **Design Patterns for Cloud Services**, *New Frontiers in Information and Software as Services*, ed. *Divyakant Agrawal, K. Selçuk Candan, Wen-Syan Li*, 31.-56., 2011.