

High-availability rešenja kod Redis baze podataka

Seminarski rad

Sistemi za upravljanje bazama podataka

Student: Uroš Pešić 1465

Profesor: Dr Aleksandar Stanimirović

Elektronski fakultet, Niš
Jun 2024.

Sadržaj

UVOD.....	3
1 – NoSQL baze podataka.....	4
1.1 – Karakteristike NoSQL baza podataka.....	5
1.2 – BASE svojstva, CAP teorema.....	7
2 – Visoka dostupnost (High-Availability).....	8
2.1 – Cena zastoja.....	8
2.2 – Uzroci narušavanja dostupnosti.....	9
2.3 – Merenje visoke dostupnosti.....	10
2.4 – Postizanje visoke dostupnosti.....	11
3 – Redis High-availability.....	13
3.1 – Redis replikacija.....	13
3.1.1 – Kako replikacija funkcioniše.....	16
3.1.2 – Read-Only replike.....	16
3.1.3 – Konfiguracija replikacije.....	17
3.2 – Automatski Failover – Redis Sentinel.....	19
3.2.1 – Konfiguracija Sentinela.....	22
3.2.2 – Primer robusne arhitekture korišćenjem Sentinela.....	24
3.2.3 – Pretplata na obaveštenja Sentinela.....	26
3.2.4 – TILT mod.....	27
ZAKLJUČAK.....	29
LITERATURA.....	30

UVOD

U današnje vreme se na internetu svakodnevno generiše ogromna količina podataka. Veliki deo tih podataka čine i nestruktuirani i polustruktuirani podaci, koji se ne uklapaju u relacione modele. Tradicionalne relacione baze podataka, pored svojih brojnih prednosti, često ne mogu da odgovore na zahteve za skalabilnošću, fleksibilnošću, dostupnošću i performansama koji se javljaju u savremenim sistemima koji skladište i obrađuju ovakve podatke. Kao odgovor na takve izazove, razvile su se NoSQL baze podataka, koje su umnogome olakšale upravljanje podacima u savremenim web sistemima.

Visoka dostupnost (*eng. High Availability*) jeste jedna od ključnih aspekata modernih sistema baza podataka. Za savremene softverske sisteme, vreme zastoja od par minuta u toku dana, ili nedostupnost 0.01% podataka, može da predstavlja velike finansijske gubitke. Zbog toga je visoka dostupnost aktivna tema istraživanja u svetu NoSQL baza podataka – njen glavni cilj jeste minimizacija gubitaka usled nepredviđenih okolnosti, ali i brza reakcija sistema u tim situacijama. Ideja ovog seminarskog rada je apostrofiranje važnosti High Availability rešenja u NoSQL bazama podataka sa posebnim osvrtom na Redis. Pregledom različitih tehnika i arhitektura koje Redis koristi za postizanje visoke dostupnosti, biće demonstrirano kako one doprinose boljoj pouzdanosti i efikasnosti celokupnog sistema i zašto su od ključnog značaja u modernim aplikacijama.

U prvom delu rada, data je potrebna teorijska osnova NoSQL baza podataka i samog koncepta visoke dostupnosti i kako se ona postiže. Akcenat u drugom delu rada je dat praktičnoj implementaciji ovih ideja kod Redis baze podataka.

1 – NoSQL baze podataka

Razvojem i rastom popularnosti web aplikacija, nedostaci SQL baza podataka u ovakvim sistemima su brzo postali evidentni. ACID (*Atomicity, Consistency, Isolation, Durability*) svojstva za izvršavanje transakcija koje SQL baze moraju da garantuju, postojanje striktnih šema za relacije su bila suviše restriktivna ograničenja, koja sa druge strane nisu bila neophodna za potrebe najvećeg dela web aplikacija. Iz ovih razloga su se razvila **NoSQL (Not-Only-SQL)** baze podataka, kao specijalizovana rešenja za skladištenje rad sa podacima u pomenutim sistemima.

Bitna karakteristika NoSQL baza podataka je **fleksibilnost šeme podataka**, što značajno olakšava razvoj aplikacija i manipulaciju podacima koji nemaju definisanu strukturu – upravo veliku količinu podataka koja se svakodnevno generiše na internetu i čuva u NoSQL bazama najvećim delom čine nestruktuirani (e-mailovi, video zapisi, fotografije, itd.) ili polustruktuirani (html stranice, podaci sa IoT senzora, itd.). U zavisnosti od modela podataka postoje sledeći tipovi NoSQL baza podataka:

- **Key-Value stores** – Ovaj tip NoSQL baza pamti podatke u obliku **ključ-vrednost**, gde je ključ string na osnovu kojeg se pristupa vrednošću koja je za njega vezana, dok vrednost može predstavljati proizvoljni tip podataka – može biti „blob“, string, celobrojna vrednost, ili složenija struktura kao što je JSON. Zbog nestruktuirane vrednosti, nije moguće izvršavati kompleksne upite slične SQL upitima, niti vršiti pretrage po sadržaju vrednosti, već samo čitanje na osnovu ključa. Pogodne su za korišćenje u sistemima u kojima se kontinualno izvršava veliki broj operacija čitanja i upisa male količine podataka. Zahvaljujući ključu i heširanju, performanse upita čitanja i upisa su značajno ubrzane. **Redis** baza podataka koja je i tema ovog rada jeste primer Key-Value NoSQL baze.
- **Document stores** – Document baze podataka su slične Key-Value bazama, u smislu da čuvaju podatke u obliku ključ-vrednost, s tim što vrednost predstavlja polustruktuirani dokument, nalik JSON-u. Dokument se sastoji iz ključ-vrednost parova koji se mogu i ugnježdavati. Takođe, u okviru jedne šeme, dokumenti mogu imati različitu strukturu. Najpoznatiji predstavnik ovog tipa NoSQL baza je MongoDB.
- **Wide-column stores** – Podaci se čuvaju u tabelama, koje imaju vrste i kolone. Međutim, za razliku od SQL baza, podaci iz različitih vrsta jedne tabele mogu imati različite kolone. Mogu se posmatrati kao dvodimenzionalna implementacija key-value baze, tako da je i pretraga vrednosti za neku kolonu jako brza. Predstavnici ovog tipa NoSQL baza su Cassandra i HBase.
- **Graf baze** – Specijalni tip NoSQL baza, prilagođene za rad sa povezanim podacima. Podaci se pamte u obliku čvorova i potega (veza). Svaki čvor modeluje neki objekat sa njegovim atributima, dok se potezima modeluju veze između objekata, koje takođe mogu imati attribute. Optimizovane su za izvršavanje upita koji predstavljaju neki vid obilaska grafa. Najpoznatiji primer ovakve NoSQL baze je Neo4J.

Postoje i NoSQL baze koje predstavljaju kombinaciju prethodno pomenutih tipova.

Takođe, sistemi koji koriste NoSQL baze za skladištenje podataka, u najvećem broju slučajeva ne moraju da zadovolje ACID svojstva koja su karakteristična za tradicionalne SQL baze, već daju prioritet boljim performansama i većoj fleksibilnosti sistema, po ceni slabijih transakcionih ograničenja. Uzmimo za primer neku društvenu mrežu koja ima veliki broj korisnika širom sveta. Za uspešan rad ovakve aplikacije nije neophodno da sistem u svakom trenutku bude potpuno konzistentan – možda neće svi korisnici u istom trenutku moći da vide najnovije objave i komentare, međutim ovo ne predstavlja problem ukoliko *eventualno* (u nekom doglednom vremenskom periodu) sistem pređe u stabilno stanje. Sa druge strane, mnogo je ozbiljniji problem ukoliko deo korisnika ne bude mogao da pristupi sistemu u periodima povećane aktivnosti, ili npr. ukoliko se performanse sistema loše i potrebno je dugo čekati na odgovor servera. Zbog toga je za ovakve i slične sisteme karakteristično odstupanje od strogih ACID svojstava, kako bi se zadovoljile pomenute karakteristike koje su detaljnije opisane u narednom poglavlju.

1.1 – Karakteristike NoSQL baza podataka

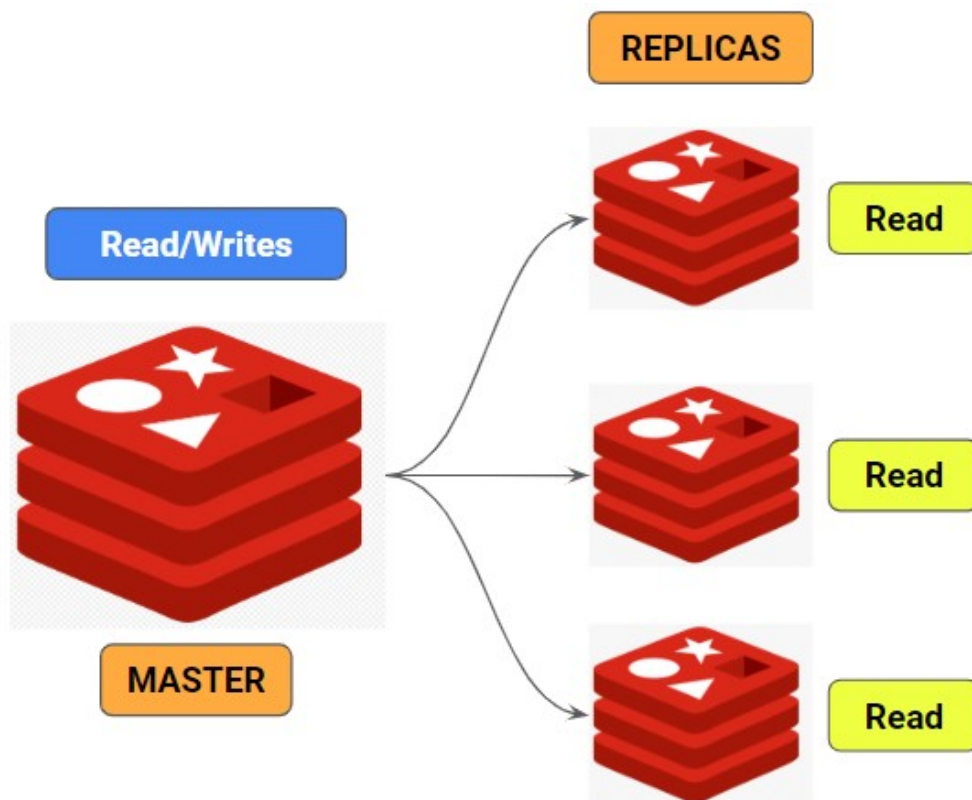
U prethodnom poglavlju je fleksibilnost šeme navedena kao bitna karakteristika NoSQL sistema. U ovom poglavlju su definisane karakteristike koje se odnose na performanse ovih sistema.

- **Skalabilnost** – NoSQL sistemi najčešće čuvaju ogromnu količinu podataka, koja svakodnevno raste. Kako bi se odgovorilo ovom izazovu, NoSQL sistemi su koriste **horizontalno skaliranje**, tj. **distribuirani** su na više čvorova u klasteru. Da bi sistem bio u mogućnosti da stalno prihvata nove generisane podatke, potrebno je obezbediti mehanizam za efikasno dodavanje čvorova u postojeći klaster, bez potrebe da se privremeno zaustavi rad sistema.
- **Dostupnost (availability), replikacija i eventualna konzistentnost** – Ove tri osobine se navode zajedno jer su međusobno usko povezane. Mnoge aplikacije koje koriste NoSQL baze po prirodi zahtevaju da podaci budu konstantno dostupni korisnicima. Da bi se to postiglo, potrebno je imati dva ili više čvorova, tako da svi sadrže iste replikovane podatke, kako bi u slučaju otkaza rezervni čvor mogao da usluži potrebe korisnika. Osim što poboljšava dostupnost, replikacija doprinosi i boljim performansama čitanja, jer je moguće balansirati opterećenje sistema i omogućiti paralelno čitanje iz različitih replika istovremeno. Sa druge strane, ovakvim rešenjem se uvodi problem nekonzistentnosti. Bilo kakva modifikacija podataka na jednoj od replika, mora biti preneti i na ostale. Ukoliko to nije moguće odmah uraditi (usled zauzetosti replike zbog druge operacije, mrežnog otkaza i slično) sistem će privremeno neće biti u konzistentnom stanju – različite replike će imati različite podatke. Zbog navedenog su i performanse operacija upisa narušene.

Postoje dva modela replikacije kojima se obezbeđuje bolja dostupnost: **master-slave** i **master-master** replikacija. Kod master-slave replikacije, postoji jedna replika koja je označena kao glavna (master) i sve operacije upisa se izvršavaju na ovoj replici, a zatim propagiraju i na ostale, tako da se eventualno postigne konzistentnost svih replika. Za

operacije čitanja postoje dva različita pristupa: prvi slučaj je prethodno opisan – čitanje je moguće sa bilo koje replike, tako da se postižu bolje performanse čitanja (mada ne postoji garancija da će pročitani podaci biti konzistentni), dok je u drugom slučaju čitanje takođe moguće samo sa master replike – tada ostale replike služe samo kao kopije sistema koje se koriste u slučaju da dođe do otkaza master replike (u tom slučaju se jedna od slave replika proglašava kao master). Master-master replikacija omogućava upis/čitanje u/iz bilo koje replike (sve su ravnopravne). Međutim, u ovom slučaju se očigledno javlja problem privremene nekonzistentnosti i dodatno je potrebno implementirati mehanizam za razrešavanje konflikata koji mogu nastati usled više operacija modifikacije istih podataka na različitim replikama. Očigledno, svaki navedeni pristup ima svoje prednosti i mane i najbolje rešenje zavisi od potrebna konkretnog sistema. Na slici 1 prikazan je primer Redis replikacije sa jednim masterom i 3 slave read-only replike (prva opisana varijanta).

- **Horizontalno particionisanje fajlova** – Kako fajlovi sa podacima u NoSQL sistemima mogu biti ogromni, vrši se njihovo horizontalno particionisanje (ili rasparčavanje – *eng. Sharding*) na više čvorova u klasteru. Podaci su najčešće particionisani na osnovu heširane vrednosti ključa. Particionisanje u kombinaciji sa replikacijom svake particije doprinosi boljem balansiranju opterećenja i boljoj dostupnosti podataka. Kako se podacima najčešće pristupa na osnovu ključa (a u slučaju key-value baza isključivo tako) particionisanje i heširanje obezbeđuju vrlo efikasan način za pretragu odgovarajuće vrste (ili više njih) među milionima slogova koji se čuvaju u bazi.



Slika 1: Redis replikacija - Jedan master sa 3 read-only slave replike

1.2 - BASE svojstva, CAP teorema

Akcent u NoSQL bazama je dat performansama, skalabilnosti i dostupnosti podataka, dok su koncepti kao što su transakcije i apsolutna konzistentost od manjeg značaja. Kako bi se adekvatno oslikale ove osobine, bilo je potrebno definisati model koji bi predstavljao alternativu ACID svojstvima tradicionalnih relacionih sistema. NoSQL sistemi su tako dobili svoj pandan ACID modelu - **BASE** model (**Basically Available, Soft State, Eventually Consistent**). Svako svojstvo će biti kratko definisano u nastavku:

- **Basically Available** – Potencira se da podaci za operacije čitanja i upisa budu dostupni što je više moguće. Da bi se to postiglo koriste se replikacija na više čvorova u distribuiranom klasteru. Očigledna posledica ovakvog pristupa je ta da replike neće biti međusobno konzistentne sve vreme, ali je ovakvo ponašanje prihvatljivo – akcent je na dostupnosti!
- **Soft State** – Za razliku od relacionih baza, koje nakon svake transakcije prelaze iz jednog u drugo jasno definisano, postojano stanje, NoSQL baze se često mogu naći u „prelaznom“ stanju, usled nepostojanja potpune konzistentnosti. Na primer, usled modifikacije istih podataka na različitim replikama, sistem može postepeno da razreši konflikte i ažurira podatke na odgovarajući način. Dok se ne finalizuje konačna vrednost modifikovanih podataka, sistem se nalazi u nekom „međustanju“.
- **Eventually Consistent**¹ – Zbog postojanja većeg broja replika, nakon modifikacije podataka je potrebno određeno vreme kako bi se te promene propagirale do svih kopija u klasteru. Za ovaj proces je potrebno određeno vreme, u toku kojeg je i dalje moguće čitanje modifikovanih podataka. Međutim, zbog privremene nekonzistentnosti ne postoji garancija da će pročitani podaci oslikavati poslednje aktuelno stanje. Činjenica da sistem *eventualno* prelazi u konzistentno stanje, nakon što se sve promene propagiraju do svih replika je upravo oslikana ovim svojstvom.

U idealnim uslovima, želeli bismo da svaka distribuirana NoSQL baza podataka ispunjava sledeća 3 svojstva:

- **Konzistentnost** (*eng. Consistency*) – Odnosi se na konzistentnost između replika, tj. na situaciju kada sve kopije sve vreme imaju identične podatke.
- **Dostupnost** (*eng. Availability*) – Sistem uvek može da prihvati operaciju čitanja ili upisa.
- **Tolerantnost na otkaze čvorova** (*eng. Partition Tolerance*) – Ukoliko čvor u sistemu otkaze, sistem ne bi trebalo da nastavi da funkcioniše neometano.

Na osnovu prethodnih poglavlja, a i na osnovu gore navedenih BASE svojstava, očigledno je da ovakav idealan sistem nije moguće realizovati. U teoriji baza podataka je ovo definisano **CAP**

¹ U distribuiranim NoSQL sistemima je moguće ostvariti i potpunu konzistentnost, međutim, zbog ogromnog overhead-a koji ovakva implementacija unosi, kojim se narušavaju pomenuta željene performanse i dostupnost koji predstavljaju suštinu ovakvih distribuiranih sistema, ovakav pristup se izbegava, osim ukoliko je neophodan.

teoremom (CAP principom), čiji je autor naučnik Eric Brewer. Prema ovoj teoremi, distribuirana NoSQL baza podataka u jednom trenutku može da ispuni najviše dve od tri navedene osobine. U praksi se najčešće preferira dostupnost i tolerantnost na otkaze, po ceni smanjene konzistentnosti (shodno 3. BASE svojstvu).

2 – Visoka dostupnost (High-Availability)

Generalno, visoka dostupnost predstavlja kontinuirani rad sistema, uz minimalno vreme zastoja, čak i u slučajevima hardverskih ili softverskih otkaza ili bilo kakvih drugih nepredviđenih situacija. Visoko dostupni sistem mora da funkcioniše gotovo bez prekida, svakog sata u danu, svakog dana u nedelji, tokom čitave godine (često se za ovu osobinu koristi izraz da sistem mora da funkcioniše 24x365). Naravno, u praksi je ovo jako teško postići i gotovo da ne postoje sistemi koji su dostupni 100% vremena. Neke od karakteristika visoko dostupnih sistema su sledeće:

- **Pouzdanost** – Visoko dostupan sistem mora da bude sačinjen od pouzdanih hardverskih i softverskih komponenti.
- **Mogućnost oporavka** – U visoko dostupnim sistemima greške se dešavaju. Ključno je unapred detektovati koje su moguće greške koje se mogu javiti i definisati najefikasniji načini oporavka od tih grešaka, kako bi se zadovoljili kriterijumi visoke dostupnosti.
- **Blagovremena detekcija grešaka** – Pored brze reakcije i oporavka usled nastanka grešaka, potrebno je otkriti uzrok te greške u nekom doglednom vremenskom intervalu, kako bi se sprečili ponovni otkazi. Zbog toga monitoring svih komponenti u sistemu predstavlja još jedan od ključnih procesa u visoko dostupnim rešenjima.
- **Kontinuirani rad** – Kao što je već pomenuto greške su često neminovne. Potrebno je obezbediti mogućnost da i pored ovakvih situacija korisnici mogu obavljati operacije upisa i čitanja podataka. Čak i u slučajevima kada je potrebno vršiti održavanje, u visoko dostupnim sistemima, ovaj proces za korisnika mora biti potpuno transparentan.

2.1 – Cena zastoja

Da bi se jasnije naznačila važnost visoke dostupnosti sistema, potrebno je kvantifikovati troškove koji se javljaju usled njenog narušavanja. Ukupnu cenu zastoja baze podataka je teško proceniti. Pored direktnih finansijskih gubitaka usled nemogućnosti pružanja usluge, potrebno je uzeti u obzir i potencijalne dugoročne posledice do kojih može doći, kao što su gubitak prednosti na tržištu u odnosu na konkurente, narušavanje poverenja i gubitak korisnika, loš publicitet i slično. Gubitak produktivnosti timova koji su zaduženi za oporavak sistema nakon zastoja se takođe ne sme zanemariti – IT timova koji su delegirani za rešavanje problema, PR menadžera i ostalih.

Iako nije moguće precizno odrediti cenu zastoja sistema, koja pored navedenih činjenica zavisi i od tipa sistema (npr. zastoj finansijskih sistema je značajno skuplji od pada sistema online edukacije), kao i od obima kompanije, na osnovu istraživanja koje je 2016. godine sproveo *Ponemon* institut, prosečna cena troškova usled neplaniranih zastoja u datacentrima je procenjena na 9000\$ po minutu, za veće kompanije[8].

2.2 – Uzroci narušavanja dostupnosti

Za uspešnu implementaciju visoko dostupnih sistema, potrebno je utvrditi potencijalne uzroke koji mogu dovesti do zastoja. Zastoji mogu biti **planirani** i **neplanirani**. Neki od uzroka neplaniranih zastoja su:

- **Kvar na lokaciji (datacentru)** – Otkaz u datacentru koji utiče na sve, ili deo procesa koji se u njemu izvršavaju. Neki od primera otkaza na lokaciji su: nestanak struje, otkaz mreže u čitavom datacentru, kvarovi usled prirodnih nepogoda, itd.
- **Otkaz klastera** – Predstavlja otkaz čitavog klastera baze podataka. Može nastupiti usled otkaza čvorova (kada i poslednji čvor otkáže i ne postoji novi koji bi preuzeo podatke i pristigle zahteve), ili usled otkaza drugih komponenti usled čega klaster postaje nedostupan.
- **Otkaz računara** – Predstavlja otkaz jednog čvora na kojem se baza izvršava, usled hardverskih otkaza, otkaza operativnog sistema, ili otkaza instance baze. Otkaz računara predstavlja manje kritičan problem od prethodna dva, jer u dobro projektovanom klasteru, u najgorem slučaju, može dovesti do nedostupnosti dela podataka, a najčešće ni do toga – ukoliko su podaci ispravno replicirani na više čvorova.
- **Mrežni otkazi** – Predstavljaju otkaz neke mrežne komponente, usled čega saobraćaj od aplikacije ka instanci baze, ili od instance baze ka fizičkom skladištu biva usporen ili onemogućen.
- **Otkaz skladišta** – Otkaz fizičkog skladišta koji čuva deo, ili celokupan sadržaj baze podataka, npr. otkaz hard diska, disk kontrolera, itd.
- **Oštećenje podataka** – Otkaz usled oštećenih podataka nastaje kada zbog greške u hardverskoj, softverskoj, ili mrežnoj komponenti (kao što su greške u menadžeru particija, drajverima diskova, disk adapteru, i sl.) dođe do čitanja oštećenog bloka podataka. Blok podataka može biti fizički „oštećen“ – ukoliko baza uopšte ne može da prepozna blok usled neslaganja kontrolne sume, ili logički „oštećen“ – baza prepoznaje pročitani blok, međutim tip pročitanih podataka se ne podudara. Posledice otkaza uzrokovanih ovakvim greškama mogu varirati – može otkazati manji deo sistema, do toga da čitava baza podataka postane nedostupna.
- **Izgubljeni i zalutali upisi** – Predstavljaju poseban vid oštećenja podataka koji se teže detektuje i ispravlja. Do ovakvih grešaka dolazi kada I/O podsistem označi neku operaciju upisa kao uspešnu, pre nego što upisani podaci budu smešteni na disk (izgubljeni upisi), ili

kada se operacija upisa obavi na nekom drugom čvoru (zalutali upisi). U oba slučaja, ukoliko nakon operacije upisa probamo da pročitamo upisane podatke, pročitane vrednosti će biti stare, neažurirane vrednosti, koje zatim mogu prouzrokovati dalje oštećenje podataka u bazi ukoliko ih koristimo za ažuriranje novih podataka.

- **Ljudski faktor** – Do otkaza sistema, ili nekog njegovog dela, može doći usled nenamerne ljudske greške ili malicioznih aktivnosti. Neki od primera su: slučajno brisanje dela podataka, nenamerno ili namerna manipulacija podacima, itd.
- **Kašnjenje, usporenje sistema** – Do ovakvih posledica dolazi najčešće usled neefikasnog upravljanja resursima, ili usled nedovoljnog kapaciteta sistema, koji ne može da odgovori na količinu zahteva koji pristižu.

U planirane zastoje spadaju:

- **Promene softvera** – Usled ispravki grešaka, bagova, ili ažuriranja sistema radi implementacije novih funkcionalnosti.
- **Promene na nivou sistema ili baze** – Odnose se na promenu hardverskih komponenti (zamena defektnih komponenti, dodavanje hardverskih resursa sistemu, itd.), promenu konfiguracionih parametara baze podataka, migracija sistema na novu arhitekturu, dodavanje čvorova klasteru, i slično.
- **Promene podataka** – Promena logičke organizacije baze, najčešće radi poboljšanja performansi sistema. Neki od primera su: promena načina particionisanja podataka, promena indeksnih struktura, itd.
- **Promene aplikacije** – Podrazumeva redovna planirana održavanja i ažuriranje aplikacije, radi poboljšanja performansi, ili ispravke grešaka.

Iako su neplanirani zastoji problematičniji, te se prilikom implementacije visoko dostupnih sistema njima posvećuje više pažnje, ne smeju se zanemariti gubici koji su prisutni usled planiranih zastoja, naročito u slučaju sistema čiji korisnici pripadaju različitim vremenskim zonama.

2.3 – Merenje visoke dostupnosti

Visoka dostupnost ne garantuje neometani rad sistema 100% vremena, već teži da taj procenat bude što bliži ovoj vrednosti. Za merenje nivoa dostupnosti koristi se sistem „devetki“. Nivo od pet devetki (**five-nines**) jeste današnji standard koji visoko dostupni sistemi moraju da ispune, i on označava dostupnost sistema tokom 99.999% vremena. Naravno, postoje pojedini sistemi koji mogu da tolerišu nešto veće vreme zastoja, bez značajnijih gubitaka. Na slici 2 prikazani su različiti nivoi dostupnosti.

Availability %	Downtime per year	Downtime per month	Downtime per week	Downtime per day
99% ("two nines")	3.65 days	7.31 hours	1.68 hours	14.40 minutes
99.5% ("two nines five")	1.83 days	3.65 hours	50.40 minutes	7.20 minutes
99.9% ("three nines")	8.77 hours	43.83 minutes	10.08 minutes	1.44 minutes
99.95% ("three nines five")	4.38 hours	21.92 minutes	5.04 minutes	43.20 seconds
99.99% ("four nines")	52.60 minutes	4.38 minutes	1.01 minutes	8.64 seconds
99.995% ("four nines five")	26.30 minutes	2.19 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	5.26 minutes	26.30 seconds	6.05 seconds	864.00 milliseconds

Slika 2: Sistem "devetki" - Različiti nivoi dostupnosti

Osim ovakve klasifikacije, definisane su još neke metrike za utvrđivanje pouzdanosti i dostupnosti sistema baza podataka:

- **Prosečno vreme između dva otkaza (MTBF)** – količnik ukupnog vremena tokom kojeg je sistem bio pušten u rad i broja kvarova koji su se u tom vremenskom periodu dogodili.
- **Prosečno vreme zastoja (MDT)** – prosečno vreme tokom kojeg sistem nije bio dostupan, bilo usled planiranih ili neplaniranih zastoja.
- **Ciljno vreme oporavka (RTO)** – predstavlja maksimalni vremenski period tokom kojeg sistem može biti u zastoju, bez da načini osetne posledice poslovnim operacijama.
- **Ciljna tačka oporavka (RPO)** – tačka u vremenu, do koje kompanija mora da povрати svoje podatke u situacijama otkaza. RPO od jednog sata, znači da kompanija može da toleriše gubitak količine podataka koji se generišu u periodu od 1h.

Različite kompanije, u zavisnosti od svojih potreba, definišu adekvatne vrednosti ovih parametara, kako bi njihovi visoko dostupni sistemi bili implementirani tako da ispunjavaju neophodne kriterijume za neometano poslovanje.

2.4 – Postizanje visoke dostupnosti

Nakon definisanja karakteristika visoko dostupnih sistema i metrika na osnovu kojih je moguće bolje proceniti njihovu uspešnost, u ovom poglavlju će biti izdvojeni ključni principi i obrasci koje je potrebno ispoštovati da bismo implementirali jedan ovakav sistem.

1. Redundantnost

Kritične komponente, čiji bi otkaz izazvao zastoj čitavog sistema je potrebno multiplicirati, kako bi u slučaju otkaza neke od njih, njena druga instanca preuzela njene odgovornosti. Na taj način se eliminišu **jedinstvene tačke otkaza** (eng. *Single Point of Failure*). U praksi se ovo postiže pomenutim tehnikama replikacije i distribucijom podataka po klasteru.

Takođe, idealno je da se se prilikom replikacije poštuje **shared-nothing** arhitektura i ovakav tip klastera Redis upravo i koristi. Kod shared-nothing arhitekture, svi čvorovi u klasteru mogu da funkcionišu nezavisno, jer ne postoje resursi koje oni dele. Zahvaljujući ovakvom visokom stepenu izolacije je moguće lakše postići visoku dostupnost – otkaze bilo koje komponente nema uticaj na ostatak sistema. Ovakvim pristupom se i olakšava skaliranje sistema usled povećanog opterećenja. Izolovane komponente je lako skalirati, jer nema potrebe voditi računa o konfliktima prilikom pristupa zajedničkim resursima, prevelikom opterećenju takvih resursa, itd.

2. **Detekcija otkaza** – Visoko dostupni sistemi moraju imati podsistem zadužen za monitoring svih ostalih komponenti, kako bi se otkaz, ukoliko do njega dođe, detektovao što ranije i obezbedio automatski failover.
3. **Automatski failover** – Kada smo se postarali za redundantnost kritičnih delova sistema, potrebno je obezbediti mehanizam koji bi u slučaju otkaza neke komponente, automatski prebacio njenu odgovornost na rezervnu repliku te komponente. Na taj način se obezbeđuje minimalno vreme zastoja u slučaju otkaza u sistemu. Na primer, kod Redis baze podataka, kada u Redis Enterprise klasteru dođe do otkaza primarne replike, neka od sekundarnih replika preuzima njenu ulogu. O ovome će biti više reči u poglavlju 3.
4. **Balansiranje opterećenja** - Replikacijom se, osim redundantnosti, postiže i mogućnost boljeg balansiranja opterećenja čitavog sistema. U slučaju da imamo više replika baze podataka, moguće je koristiti bilo koju od njih za usluživanje operacije čitanja podataka. Posebna komponenta sistema – **load balancer** tada može prosleđivati zahteve ka odgovarajućoj replici, shodno njihovim trenutnim opterećenjima. Ovaj aspekt je ključan za održanje zadovoljavajućih performansi visoko dostupnih sistema.

Kako je teško predvideti sve potencijalne uzroke otkaza koji bi narušili dostupnost sistema, u praksi se često primenjuje **inženjering haosa**. Ova tehnika podrazumeva namerno uvođenje grešaka u softverski sistem u produkciji, kako bi se otkrile slabe tačke sistema i kreirala rešenja za otkrivene probleme. Primenom inženjeringa haosa u produkcionom okruženju, moguće je uvideti realne probleme koji se mogu javiti, koje bi bilo teško otkriti u simuliranim okruženjima.

3 – Redis High-availability

U prethodnom poglavlju su definisane neke teorijske osnove i generalni obrasci za postizanje visoke dostupnosti. Cilj ovog poglavlja je da kroz primere demonstrira primenu nekih pomenutih tehnika kod Redis baze podataka.

Redis (Remote Dictionary Server) je distribuirana key-value NoSQL baza podataka. Specifična je po tome što podatke čuva u glavnoj memoriji (in-memory store) i zahvaljujući toj činjenici ima jako dobre performanse upisa i čitanja. Opciono je moguće omogućiti i perzistenciju na disk, kako ne bi došlo do gubitka podataka u slučaju otkaza, ili restartovanja čvora na kome se izvršava. Zbog ovih svojih osobina se često koristi za potrebe keširanja podataka, implementaciju real-time rang listi, redova poruka, a zahvaljujući svom *publish/subscribe* mehanizmu i za real-time razmenu poruka. Danas, zahvaljujući svojim odličnim performansama u navedenim domenima, kao i funkcionalnostima koje omogućavaju postizanje „five-nines“ nivoa dostupnosti, Redis predstavlja jednu od najpopularnijih NoSQL baza podataka, koju koriste i kompanije kao što su Amazon, Twitter, Airbnb, i mnoge druge.

Za postizanje visoke dostupnosti, Redis koristi tehnike **replikacije** (kako *Active/Passive*, tako i *Active/Active* replikacije u enterprise varijanti), **automatski failover** (korišćenjem *Redis Sentinel*-a), itd. Ove tehnike biće objašnjene u nastavku.

3.1 – Redis replikacija

Redis podržava *Leader-Follower (Master-Slave)* replikaciju (kao što je prikazano na slici 1), out-of-the-box koja je jednostavna za konfiguraciju. Svaka replika predstavlja identičnu kopiju master instance. Ukoliko dođe do prekida veze između mastera i replika, replike će automatski probati da se povežu nazad na master instancu i, ukoliko to uspešno učini, dobiće instrukcije kako bi prihvatile nove podatke koji su u međuvremenu bili upisani na masteru. Ovo se postiže zahvaljujući sledećim mehanizmima:

- Dok postoji veza između mastera i replika, sve operacije koje modifikuju podatke na bilo koji način (bilo da je reč o operacijama koje klijent izvršava, brisanju podataka usled isteka ključeva, itd) master šalje replikama.
- Ukoliko dođe do prekida konekcije, replika se ponovo povezuje na master instancu i pokušava da izvrši *parcijalnu resinhronizaciju*- tj. zahteva od mastera joj posalje tok komandi koje je propustila, kako bi ponovo dobila aktuelne podatke koji su u međuvremenu promenjeni.
- Ukoliko parcijalna sinhronizacija ne bude uspešna, replika od master instance zahteva *potpunu resinhronizaciju*. U ove svrhe master instanca pravi *snapshot* čitave baze i šalje ih replici, kako bi se ona ponovo sinhronizovala, a zatim nastavlja sa slanjem toka komandi,

kako bi se konzistentnost održala. Naravno potpuna sinhronizacija je vremenski zahtevniji proces.

Na sledećim slikama su ilustrovani procesi potpune i parcijalne resinhronizacije.

[illegible]

Slika 3: Replikacija - Potpuna resinhronizacija

Na slici iznad možemo da vidimo kako izgleda proces potpune resinhronizacije. Predstavljeno je kreiranje nove replike na osnovu postojećeg mastera. Kako je nova replika tek kreirana, potrebno je da master instanca njoj dostavi snapshot tekućeg stanja baze (RDB fajl), što možemo da vidimo na osnovu logova. Na kraju procesa, replika će sadržati ista 3 ključa kao i master, što znači da je replikacija i sinhronizacija uspešna.

```
4693:M 08 Jul 2024 20:39:31.644 * Connection with replica 127.0.0.1:6380 lost.
4693:M 08 Jul 2024 20:39:39.675 * Replica 127.0.0.1:6380 asks for synchronization
4693:M 08 Jul 2024 20:39:39.675 * Partial resynchronization request from 127.0.0.1:6380 accepted. Sending 80 bytes of backlog starting from offset 2412.

MASTER

Redis 7.2.5 (00000000/0) 64 bit
Running in standalone mode
Port: 6380
PID: 5907
https://redis.io

REPLICA

5907:S 08 Jul 2024 20:39:39.666 * Server initialized
5907:S 08 Jul 2024 20:39:39.674 * Loading RDB produced by version 7.2.5
5907:S 08 Jul 2024 20:39:39.674 * RDB age 8 seconds
5907:S 08 Jul 2024 20:39:39.674 * RDB memory usage when created 1.28 Mb
5907:S 08 Jul 2024 20:39:39.674 * Done loading RDB, keys loaded: 3, keys expired: 0.
5907:S 08 Jul 2024 20:39:39.674 * DB loaded from disk: 0.000 seconds
5907:S 08 Jul 2024 20:39:39.674 * Before turning into a replica, using my own master parameters to synthesize a cached master: I may be able to synchronize with the new master with just a partial transfer.
5907:S 08 Jul 2024 20:39:39.674 * Ready to accept connections tcp
5907:S 08 Jul 2024 20:39:39.674 * Connecting to MASTER 127.0.0.1:6379
5907:S 08 Jul 2024 20:39:39.674 * MASTER <-> REPLICA sync started
5907:S 08 Jul 2024 20:39:39.674 * Non blocking connect for SYNC fired the event.
5907:S 08 Jul 2024 20:39:39.675 * Master replied to PING, replication can continue...
5907:S 08 Jul 2024 20:39:39.675 * Trying a partial resynchronization (request b5b17283881a4caebc7bb7ceb4513d0a9dc1b2:2412).
5907:S 08 Jul 2024 20:39:39.675 * Successful partial resynchronization with master.
5907:S 08 Jul 2024 20:39:39.675 * MASTER <-> REPLICA sync: Master accepted a Partial Resynchronization.
```

Slika 4: Replikacija - Parcijalna sinhronizacija

Na slici 4 je simuliran prekid konekcije između mastera i replike, tako što je replika privremeno ugašena. Nakon gašenja, na master instanci su dodati novi podaci korišćenjem komande operacije SET, a zatim je replika ponovo upaljena. Kao što možemo da vidimo sa slike, u ovoj situaciji master inicira parcijalnu resinhronizaciju i šalje replici komande koje je propustila. Nakon izvršenja ovih komandi replika ponovo predstavlja verodostojnu kopiju mastera.

Redis podrazumevano koristi asinhronu replikaciju – Nakon što se operacija uspešno izvrši na master instanci i bude prosleđena svim replikama, master se ne blokira čekajući da je i replike izvrše, već nastavlja da prima dalje zahteve. Ovakvom implementacijom se održavaju visoke performanse i minimalno kašnjenje sistema. Ukoliko je potrebno, moguće je predefinisati ovakvo ponašanje korišćenjem WAIT komande nakon prilikom zadavanja operacija. WAIT komanda ima sledeći oblik:

WAIT broj_replika timeout;

WAIT funkcioniše tako što će da blokira sistem dok broj replika (definisan prvim parametrom) nije potvrdio uspešno izvršenje emitovane operacije, ili dok ne istekne timeout milisekundi. Ovakva sinhrona logika potencira bolju konzistentnost po ceni manje dostupnosti i generalno lošijih performansi sistema, pa se izbegava u visoko dostupnim sistemima, osim ukoliko nije apsolutno neophodna.

Takođe, proces parcijalne sinhronizacije je neblokirajući i na replikama – ukoliko je u konfiguraciji instanci replika naveden parametar replica-serve-stale-data yes, tada replika može da usluđuje zahteve za čitanjem podataka iako je sinhronizacija u toku. Treba imati u vidu da ovakvom

konfiguracijom može doći do čitanja zastarelih podataka - ukoliko ne možemo da priuštimo da se to desi, onda treba isključiti ovu opciju.

3.1.1 – Kako replikacija funkcioniše

Interno, svaka Redis master instanca sadrži ID replikacije – pseudo-nasumični string koji predstavlja neki vid pečata koji označava verziju podataka koju ta instanca sadrži i pomeraj, koji se inkrementira sa svakim bajtom koji se pošalje replikama radi sinhronizacije. Pomeraj se inkrementira čak i u slučaju da nijedna replika trenutno nije povezana na master instancu. Ove dve oznake zajedno nedvosmisleno definišu trenutnu verziju podataka u svim Redis instancama.

Kada se nova replika poveže na master, ona interno, komandom PSYNC, šalje svoj ID replikacija i pomeraj, kako bi obavestila mastera do koje operacije je uspešno pratila promene u podacima. Nakon što dobije ovu vrednost (pod uslovom da se ID-evi replikacije poklapaju), master toj replici šalje sve operacije od te vrednosti pomeraja do kraja. U ovom kontekstu se pomeraj može posmatrati kao logički časovnik koji označava do kog trenutka u vremenu je tekuća instanca pratila promene u master instanci. U drugom slučaju, kada je ID replikacije replike i mastera različit, parcijalna sinhronizacija nije moguća, već master instanca u pozadinskom procesu kreira snapshot baze i inicira potpunu sinhronizaciju.

Osim primarnog ID-a replikacije, svaka master instanca sadrži i sekundarni ID replikacije. Ovo je bitno u slučajevima automatskog failover-a, kada neka od instanci mora da preuzme ulogu mastera. Tada ona dobija novi primarni ID replikacije, dok sekundarni ID replikacije predstavlja prethodno korišćeni ID mastera (ID mastera pre otkaza). Tada, u slučaju da neka od replika čiji je ID ekvivalentan sekundarnom ID-u novog mastera može da bude sinhronizovana korišćenjem parcijalne sinhronizacije, čime se postiže bolja efikasnost i brža stabilizacija sistema u slučaju otkaza master instanci. Na slikama 3 i 4 možemo da vidimo kako izgledaju ove dve anotacije posmatrajući logove u terminalu: možemo da primetimo ID replikacije mastera *b5b1728...c1b2*, i pomeraje – 17 u prvom slučaju, a zatim 2412 za vreme parcijalne sinhronizacije, jer je u međuvremenu vršeno dodavanje podataka u bazu.

3.1.2 – Read-Only replike

Za postizanje bolje dostupnosti i boljih performansi sistema u kojima su operacije čitanja značajno učestalije od operacija upisa, moguće je omogućiti čitanje iz replika navođenjem parametra *replica-read-only yes* u konfiguracionom fajlu. U slučaju read-only replika je moguće balansirati opterećenje prosleđivanjem zahteva za čitanje različitim instancama, dok će operacije čitanja biti odbijene. Primer je prikazan na slici 5.


```
>_ CLI
Connecting...

Pinging Redis server on 127.0.0.1:6380
Connected.
Ready to execute commands.

> SET 6 "can i do this?"
"READONLY You can't write against a read only replica."

> GET 5
"new data while replica is down"
```

Slika 5: Read-only replika, pokušaj upisa i čitanje

Iako je moguće omogućiti upis u repliku, postavljanjem prethodno pomenutog parametra na *no*, ovakva praksa se ne preporučuje. Upisom u replike dolazi do nekonzistentnosti baze podataka koju Redis ne može sam da ispravi – propagacija promena se vrši od mastera ka replikama, dok obrnuti smer nije moguć. Replike koje dozvoljavaju upis su prisutne iz istorijskih razloga. Postoje različite Redis operacije, kao što su *SUNIONSTORE*, *ZINTERSTORE*, *SORT*, itd. koje generisani rezultat mogu privremeno upisati u bazu. Za ovakve operacije Redis poseduje njihovu read-only alternativu, koju je moguće koristiti i nad replikama – *SUNION*, *ZINTER*, *SORT_RO*.

3.1.3 – Konfiguracija replikacije

Prilikom startovanja Redis instance, potrebno je navesti putanju do konfiguracionog fajla koji sadrži potrebne parametre (Redis prilikom instalacije dolazi uz podrazumevani *redis.conf* fajl). Za kreiranje replike, potrebno je u konfiguracionom fajlu nove instance navesti parametre:

port „broj porta“, gde se broj porta naravno mora razlikovati od broja porta master instance
replicaof „IP adresa mastera“ „port mastera“

Dodatno, moguće je specificirati još neke od pomenutih parametara:

- *replica-read-only yes* (yes je podrazumevana vrednost ovog parametra)
- *replica-serve-stale-data* – mogućnost čitanja i potencijalno zastarelih podataka iz replike. Narušava potpunu konzistentnost po ceni boljih performansi čitanja (jer je proces sinhronizacije asinhroni)
- *masterauth* „master lozinka“ – ukoliko je master instanca zaštićena lozinkom, nju je potrebno navesti prilikom startovanja replike

- `requirepass „lozinka“` – ukoliko želimo da i replika bude zaštićena lozinkom, to možemo uraditi ovim parametrom

Primer jedne proste konfiguracije je dat na sledećoj slici:

```
1 # replication
2 port 6380
3 replicaof 127.0.0.1 6379
4
5 # read-only, allow stale data
6
7 replica-read-only yes
8 replica-serve-stale-data yes
9
10 # auth to master, require auth for replica
11
12 masterauth uros2307
13 requirepass uros2307
```

Slika 6: Primer konfiguracionog fajla za repliku

Osim navedenih parametara, u nastavku su navedeni još neki parametri koji potencijalno mogu biti korisni:

- `min-replicas-to-write „broj replika“`
`min-replicas-max-lag „broj sekundi“`

Ova dva parametra se zajedno koriste kada hoćemo da zabranimo upis u master instancu, osim ukoliko na nju nije povezan definisani broj replika (što se definiše prvom navedenom prvom komandom). Takođe, te replike ne smeju da kasne više od „broj sekundi“ sekundi. Da bi vodio evidenciju o broju dostupnih replika i njihovom kašnjenju, master instanca svake sekunde prima PING signal od svih replika koje su na nju povezane.

U ovom poglavlju smo videli kako je Redis replikacijom obezbeđen prvi uslov za postizanje visoke dostupnosti – replikovanjem master instance se postiže redundantnost i poboljšavaju performanse sistema (read-only replike), tj. moguće je bolje balansiranje opterećenja. Tema narednog poglavlja jeste još jedan neizostavni aspekt visoko dostupnih sistema – automatski failover.

3.2 – Automatski Failover – Redis Sentinel

Redis Sentinel predstavlja rešenje koje obezbeđuje automatski failover za postizanje visoke dostupnosti u Redis non-cluster² sistemima. Osim ove glavne funkcionalnosti, Redis Sentinel obavlja i druge zadatke, ključne za visoko dostupne sisteme:

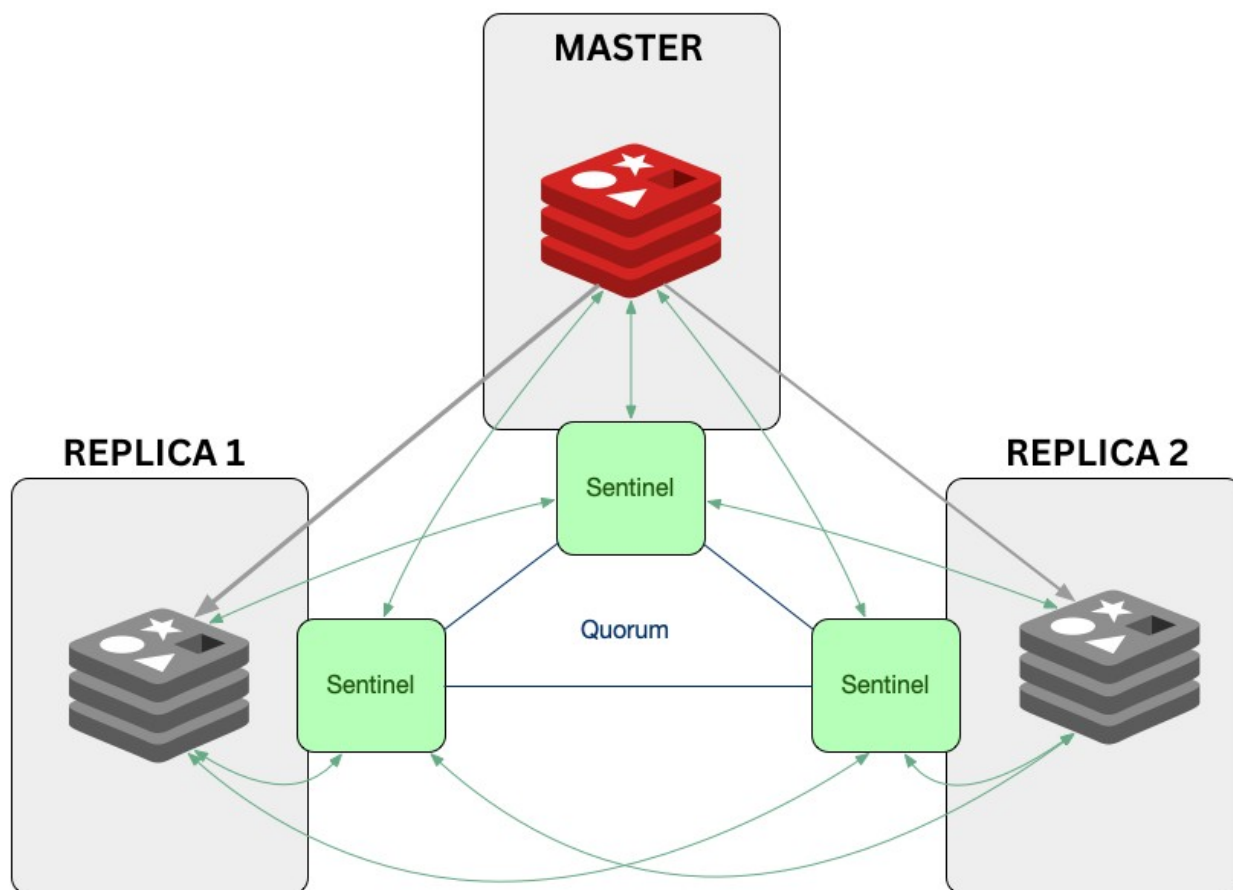
- **Monitoring** – Redis Sentinel konstantno nadgleda status master instance i replika kojima je dodeljen.
- **Obaveštavanje** – Posедуje mogućnost da notifikuje administratora, ili druge programe, kada dođe do zastoja neke od instanci koje nadgleda.
- **Provajder konfiguracije** – Klijentske aplikacije mogu da koriste Redis Sentinel za otkrivanje servisa – klijenti se prvo povezuju na instancu Sentinel-a kako bi dobili IP adresu master instance. Ukoliko usled otkaza mastera nova instanca preuzme ovu ulogu, Sentinel će o ovome obavestiti klijente.

Redis Sentinel predstavlja distribuirani sistem koji se sastoji iz više Sentinel procesa koji međusobno komuniciraju i sarađuju kako bi detektovali otkaze i obezbedili automatski failover. Ovakva implementacija ima dve bitne prednosti:

- Da bi se detektovao otkaz mastera, više instanci Sentinela mora da se složi oko ove odluke. Ovakvim pristupom se smanjuje verovatnoća pojave lažnih otkaza (na primer, kada bismo imali samo jednu komponentu koja je zadužena da detektuje otkaze, mrežni otkaz između ove komponente i master instance bi bio detektovan kao otkaz mastera – iako ovo nije slučaj)
- Ukoliko jedna instanca Sentinela otkáže, druge instance nastavljaju da nadgledaju sistem – ovo čini sistem robusnijim na otkaze. Ukoliko bi Sentinel imao samo jednu instancu, on bi tada predstavljao jedinstvenu tačku otkaza, koja ne sme da postoji u visoko dostupnim sistemima!

Redis instance (masteri i njihove replike), Redis Sentineli i klijenti koji su konektovani na bazu ili Sentinele zajedno čine jedan veliki distribuirani sistem. Primer ovakvog sistema je prikazan na slici 7.

2 Termin „cluster“ se u ovom kontekstu odnosi na činjenicu da se ne koristi Redis Cluster rešenje – više čvorova, svaki sa master slave replikama, na kojima su podaci koji su razbijeni u particije (shard-ove). Ovakav Redis Cluster se koristi za potrebe horizontalnog skaliranja radi boljih performansi i tolerantnosti na otkaz particija. Redis Sentinel i replikacija se, naravno mogu koristiti na više čvorova, radi postizanja bolje redundantnosti i samim tim visoke dostupnosti.



Slika 7: Redis Sentinel i Redis instance - Distribuirani sistem

Kvorum predstavlja jedan od ključnih parametara Redis Sentinel – kvorumom se definiše minimalan broj Sentinel instanci koje moraju da se slože da je došlo do otkaza master instance da bi se pokrenuo proces automatskog failover-a. *Kvorum se koristi samo za potrebe detekcije otkaza!* U slučaju da se Sentineli slože da je master instanca otkazala, jedan od Sentinel a će pokušati da startuje proces failover-a. Proces će se uspešno startovati samo ukoliko je većina Sentinel instanci potvrdila ovaj proces. Važno je napomenuti da ovo važi samo u slučajevima kada je kvorum manji od $(\text{broj_sentinel_instanci} / 2)$. Ukoliko je kvorum veći od ovog broja, tada je potrebno da najmanje kvorum instanci takođe potvrdi failover, kako bi se on izvršio. Ovo je jasnije ilustrovano na sledećim primerima:

```
3242:X 09 Jul 2024 18:37:28.663 * Sentinel new configuration saved on disk
3242:X 09 Jul 2024 18:37:28.664 * Sentinel ID is 0d6cf3b5595d3c0d7a3f7b5115f6d04748e711d3
3242:X 09 Jul 2024 18:37:28.664 # +monitor master myprimary 127.0.0.1 6379 quorum 2
3242:X 09 Jul 2024 18:37:28.665 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ myprimary 127.0.0.1 6379
3242:X 09 Jul 2024 18:37:28.674 * Sentinel new configuration saved on disk
3242:X 09 Jul 2024 18:38:33.181 * +sentinel sentinel e513daec9c10a4060c93ec859c9f37ba873ed09e 127.0.0.1 5001 @ myprimary 127.0.0.1 6379
3242:X 09 Jul 2024 18:38:33.192 * Sentinel new configuration saved on disk
3242:X 09 Jul 2024 18:39:24.330 # +sdown sentinel e513daec9c10a4060c93ec859c9f37ba873ed09e 127.0.0.1 5001 @ myprimary 127.0.0.1 6379
3242:X 09 Jul 2024 18:41:15.318 # +sdown master myprimary 127.0.0.1 6379
3242:X 09 Jul 2024 18:41:39.796 # -sdown master myprimary 127.0.0.1 6379
```

Slika 8: Kvorum nije postignut

Na slici 8 je prikazan izlaz na kome možemo da vidimo događaje koje Sentinel detektuje. Za potrebe demonstracije je kreirana master instanca sa jednom replikom i pokrenute su sve Sentinel instance. Kvorum je postavljen na 2. U donjem uokvirenom delu slike možemo da vidimo da je jedan sentinel otkazao (namerno je ugašen za potrebe simulacije otkaza koji bi potencijalno nastupio u realnim uslovima). To znači da sada samo jedna Sentinel instanca ne može da postigne kvorum za detekciju otkaza mastera – kao što se vidi na dnu slike Sentinel detektuje da je primarna replika otkazala (događaj `+sdown` u pretposlednjoj vrsti), međutim nema pokušaja pokretanja failover-a jer kvorum nije dostignut.

```
3791:X 09 Jul 2024 18:46:35.505 # +sdown sentinel 26b2122d901c0a983aad451ea06cd9e104dfbed 127.0.0.1 5001 @ myprimary 127.0.0.1 6379
3791:X 09 Jul 2024 18:46:53.129 # +sdown master myprimary 127.0.0.1 6379
3791:X 09 Jul 2024 18:46:53.129 # +odown master myprimary 127.0.0.1 6379 #quorum 1/1
3791:X 09 Jul 2024 18:46:53.129 # +new-epoch 1
3791:X 09 Jul 2024 18:46:53.129 # +try-failover master myprimary 127.0.0.1 6379
3791:X 09 Jul 2024 18:46:53.137 * Sentinel new configuration saved on disk
3791:X 09 Jul 2024 18:46:53.137 # +vote-for-leader 23ca5e21dd041404ea24e1704d4fbc0f9d85a8f0 1
3791:X 09 Jul 2024 18:47:00.068 # -failover-abort-not-elected master myprimary 127.0.0.1 6379
```

Slika 9: Kvorum postignut, nema failover-a

Na slici 9 je prikazan drugi primer. Postavka arhitekture je ista kao u prethodnom primeru, s tim što je sada kvorum postavljen na vrednost 1. Nakon „otkaza“ jedne instance Sentinela, druga instanca detektuje otkaz i postiže se kvorum (jer je dovoljna samo jedna instanca u ovom slučaju), nakon čega Sentinel instanca koja je detektovala otkaz pokušava da pokrene proces failover-a (3. linija na uokvirenom delu slike). Međutim, u poslednjem redu možemo da vidimo da ovaj proces nije uspeo, jer većina za izglasavanje novog mastera nije postignuta, jer je jedna od dve početne instance Sentinela otkazala.

```
4175:X 09 Jul 2024 18:54:36.987 # +sdown sentinel db8fea3db03476c55c3560d6ab591fb9a4706f70 127.0.0.1 5002 @ myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:34.839 # +sdown master myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:34.930 # +odown master myprimary 127.0.0.1 6379 #quorum 2/2
4175:X 09 Jul 2024 18:55:34.930 # +new-epoch 1
4175:X 09 Jul 2024 18:55:34.930 # +try-failover master myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:34.939 * Sentinel new configuration saved on disk
4175:X 09 Jul 2024 18:55:34.940 # +vote-for-leader f845ce60fc6b9b971d7a871e57c84c31c5aac879 1
4175:X 09 Jul 2024 18:55:34.949 * e8d572196f8ff7b5f9bc0ca63cd18842fb97d55d voted for f845ce60fc6b9b971d7a871e57c84c31c5aac879 1
4175:X 09 Jul 2024 18:55:34.993 # +elected-leader master myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:34.993 # +failover-state-select-slave master myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:35.070 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:35.070 # +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @ myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:35.126 # +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ myprimary 127.0.0.1 6379
4175:X 09 Jul 2024 18:55:35.980 * Sentinel new configuration saved on disk
```

Slika 10: Uspešni failover

Konačno, na slici 10 je proces automatskog failover-a uspeo. Za potrebe ovog primera su pokrenute 3 Sentinel instance, dok je za postizanje kvoruma potrebno da se bar 2 slože oko otkaza. Gašenjem jedne od 3 instance se simulira njen otkaz, a nakon toga se simulira otkaz master instance. Sa slike možemo da vidimo da su se preostale dve aktivne Sentinel instance složile da je master instanca otkazala (prva vrsta na uokvirenom delu slike). Nakon toga je pokušano pokretanje failover-a, koje je i uspešno jer su obe instance izglasale repliku na adresi `127.0.0.1 6380` kao novog mastera, čime je proces failover-a uspešno završen.

Navedeni primeri takođe treba da naglase i tipične probleme koji se mogu javiti ukoliko koristimo samo dve Sentinel instance. Otkazom bilo koje od njih, gubi se mogućnost automatskog failover-a, čime čitav sistem postaje nedostupan. Ovo je loša praksa koju treba izbegavati. Generalno, potrebne su bar 3 instance Sentinela za robusnu arhitekturu.

3.2.1 – Konfiguracija Sentinela

Jedna Redis Sentinel instanca se pokreće komandom **redis-server sentinel.conf --sentinel**, gde **sentinel.conf** predstavlja konfiguracioni fajl koji je neophodno navesti. U konfiguracionom fajlu se navodi svi potrebni konfiguracioni parametri. Primer najjednostavnijeg konfiguracionog fajla je prikazan na sledećoj slici:



Slika 11: Sentinel - Konfiguracioni fajl

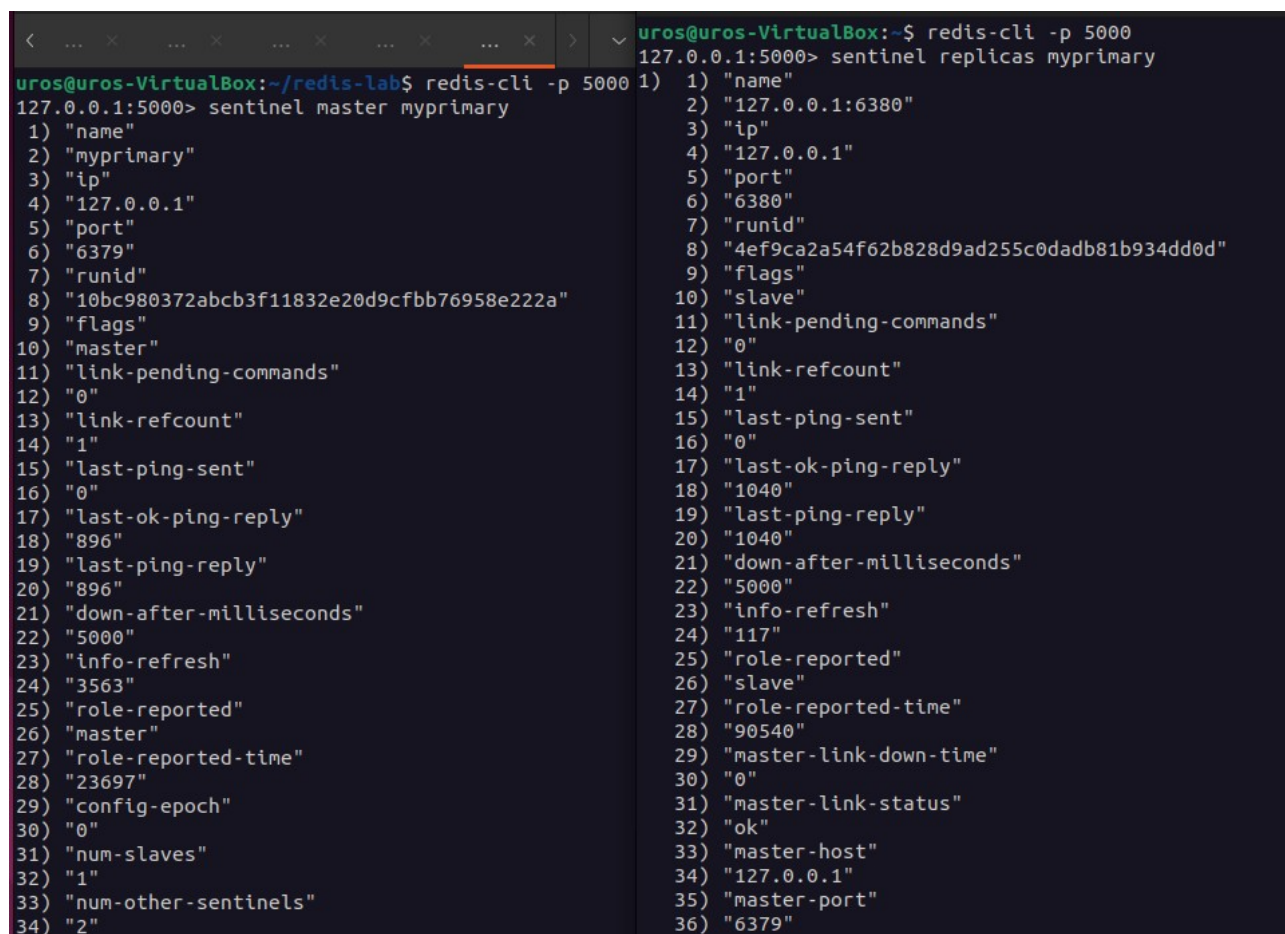
Navedene konfiguracione opcije imaju sledeće efekte:

- **sentinel monitor <ime> <IP adresa mastera> <port mastera> <quorum>** - Sentinel u se dodeljuje master koji treba da se nadgleda navođenjem njegove IP adrese i porta. Nije potrebno navoditi adrese replika, Sentinel na osnovu mastera može sam da otkrije adrese svih aktivnih replika.
- **sentinel down-after-milliseconds <ime><broj_milisekundi>** - Definiše se minimalni vremenski interval tokom kojeg neka instanca mora biti neaktivna, da bi se proglasio njen otkaz
- **sentinel failover-timeout <ime> <broj_milisekundi>** - Vremenski interval nakon kojeg će failover ponovo moći da se izvrši, nakon što je već bio izvršen.
- **sentinel auth-pass <ime> <lozinka>** - Lozinka za pristup master instanci.
- **sentinel parallel-syncs <ime> <broj>** - Definiše broj replika koje će moći istovremeno da se sinhronizuju sa novim masterom nakon failover-a. Navođenjem većeg broja se proces resinhronizacije ubrzava, međutim postoji kratak vremenski period tokom kojeg će replike koje se istovremeno sinhronizuju biti nedostupne (zbog učitavanja podataka sa mastera). Sa druge strane, ukoliko su replike konfigurisane kao read-only i imaju mogućnost da

dostavljaju i zastarele podatke, postavljanjem niže vrednosti za ovu opciju se postiže bolja dostupnost sistema (jer će u svakom trenutku bar jedna replika biti aktivna).

U donjem delu slike možemo da vidimo zbog čega je neophodno navođenje konfiguracionog fajla za Sentinele - u toku nadgledanja sistema Sentineli prepisuju, tj. dopisuju stvari koje otkriju. U konkretnom slučaju, na slici 11 možemo da vidimo da je ova Sentinel instanca otkrila i druge dve instance, na adresama 127.0.0.1:5001 i 127.0.0.1:5002.

Nakon što su Sentinel instance pokrenute, različitim komandama je moguće videti informacije o masteru, replikama, ili drugim Sentinelima, korišćenjem komande `sentinel <komanda> <ime>`. Na sledećoj slici je prikazan primer ovakvog pribavljanja informacija o masteru i replikama koje Redis Sentinel nadgleda.



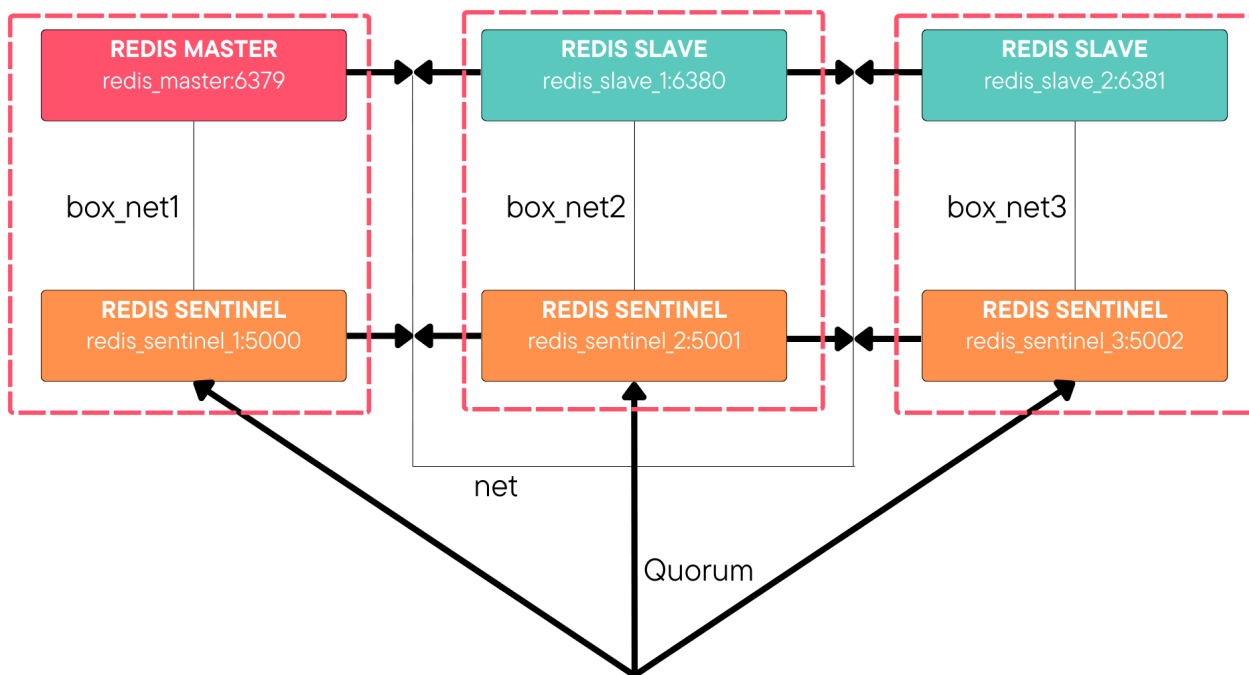
```
uros@uros-VirtualBox:~$ redis-cli -p 5000
127.0.0.1:5000> sentinel master myprimary
1) "name"
2) "myprimary"
3) "ip"
4) "127.0.0.1"
5) "port"
6) "6379"
7) "runid"
8) "10bc980372abcb3f11832e20d9cfbb76958e222a"
9) "flags"
10) "master"
11) "link-pending-commands"
12) "0"
13) "link-refcount"
14) "1"
15) "last-ping-sent"
16) "0"
17) "last-ok-ping-reply"
18) "896"
19) "last-ping-reply"
20) "896"
21) "down-after-milliseconds"
22) "5000"
23) "info-refresh"
24) "3563"
25) "role-reported"
26) "master"
27) "role-reported-time"
28) "23697"
29) "config-epoch"
30) "0"
31) "num-slaves"
32) "1"
33) "num-other-sentinels"
34) "2"

uros@uros-VirtualBox:~$ redis-cli -p 5000
127.0.0.1:5000> sentinel replicas myprimary
1) "name"
2) "127.0.0.1:6380"
3) "ip"
4) "127.0.0.1"
5) "port"
6) "6380"
7) "runid"
8) "4ef9ca2a54f62b828d9ad255c0dadb81b934dd0d"
9) "flags"
10) "slave"
11) "link-pending-commands"
12) "0"
13) "link-refcount"
14) "1"
15) "last-ping-sent"
16) "0"
17) "last-ok-ping-reply"
18) "1040"
19) "last-ping-reply"
20) "1040"
21) "down-after-milliseconds"
22) "5000"
23) "info-refresh"
24) "117"
25) "role-reported"
26) "slave"
27) "role-reported-time"
28) "90540"
29) "master-link-down-time"
30) "0"
31) "master-link-status"
32) "ok"
33) "master-host"
34) "127.0.0.1"
35) "master-port"
36) "6379"
```

Slika 12: Provera stanja sistema korišćenjem Sentinel

3.2.2 – Primer robusne arhitekture korišćenjem Sentinel

Već je na početku poglavlja 3.2 pomenuto kako bi trebalo izbegavati sisteme sa manje od tri Redis Sentinel. Takođe, pomenuto je kako je preporučljivo distribuirati replike na različite čvorove, kako otkazom jednog čvora ne bismo izgubili više Redis instanci. Isto važi i za Sentinele. Shodno tome, na slici 13 je dat primer jedne potencijalne arhitekture, koja je robusna i pogodna za visoko dostupne sisteme.



Slika 13: Primer robusne arhitekture

Ovakva arhitektura je simulirana korišćenjem Redis i Redis Sentinel docker kontejnera. Zbog toga možemo da posmatramo kao da se svaka instanca sa slike izvršava u zasebnom čvoru, mada bi ovakav scenario bio skup u realnosti, te bi za bolje iskorišćenje resursa, bilo moguće grupisati po jednu Redis repliku sa jednim Sentinel procesom (označeno crvenim isprekidanim pravougaonicima). U nastavku ćemo podrazumevati da je svaka instanca sa slike na zasebnom čvoru (docker kontejneru).

Robusnost sistema je očigledna – otkazom bilo kog čvora sistem može da nastavi da funkcioniše neometano. Primer otkaza mastera je dat u nastavku:


```
uros@uros-VirtualBox: ~/redis-lab
uros@uros-VirtualBox:~/redis-lab$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
6dcb6b8997f1   bitnami/redis-sentinel:latest       "/opt/bitnami/script..." 12 seconds ago Up 12 seconds 0.0.0.0:5002->5002/tcp, :::5002->5002/tcp, 26379/tcp redis_sentinel_3
bd0cff9a1555   bitnami/redis-sentinel:latest       "/opt/bitnami/script..." 15 seconds ago Up 14 seconds 0.0.0.0:5001->5001/tcp, :::5001->5001/tcp, 26379/tcp redis_sentinel_2
67afba231961   bitnami/redis-sentinel:latest       "/opt/bitnami/script..." 17 seconds ago Up 17 seconds 0.0.0.0:5000->5000/tcp, :::5000->5000/tcp, 26379/tcp redis_sentinel_1
bd4bb412c31a   redis                                "docker-entrypoint.s..." 20 seconds ago Up 19 seconds 6379/tcp, 0.0.0.0:6381->6381/tcp, :::6381->6381/tcp redis_slave_2
55d25a524fd2   redis                                "docker-entrypoint.s..." 22 seconds ago Up 22 seconds 6379/tcp, 0.0.0.0:6380->6380/tcp, :::6380->6380/tcp redis_slave_1
e6e6fdd5b085   redis                                "docker-entrypoint.s..." 25 seconds ago Up 24 seconds 0.0.0.0:6379->6379/tcp, :::6379->6379/tcp redis_master

uros@uros-VirtualBox:~/redis-lab$

uros@uros-VirtualBox:~$ redis-cli -p 6379
127.0.0.1:6379> GET 1
(nil)
127.0.0.1:6379> SET 1 "first"
OK
127.0.0.1:6379> GET 1
"first"
127.0.0.1:6379>

MASTER

uros@uros-VirtualBox:~$ redis-cli -p 6380
127.0.0.1:6380> GET 1
(nil)
127.0.0.1:6380> GET 1
"first"
127.0.0.1:6380> SET 2 "second"
(error) READONLY You can't write against a read only replica.
127.0.0.1:6380>

REPLICA 1

uros@uros-VirtualBox:~$ redis-cli -p 6381
127.0.0.1:6381> GET 1
(nil)
127.0.0.1:6381> GET 1
"first"
127.0.0.1:6381> SET 2 "second"
(error) READONLY You can't write against a read only replica.
127.0.0.1:6381>

REPLICA 2

uros@uros-VirtualBox:~$ redis-cli -p 6381
127.0.0.1:6381> GET 1
"641"
127.0.0.1:6381> GET 1
"last-ping-reply"
127.0.0.1:6381> GET 1
"641"
127.0.0.1:6381> GET 1
"down-after-milliseconds"
127.0.0.1:6381> GET 1
"5000"
127.0.0.1:6381> GET 1
"info-refresh"
127.0.0.1:6381> GET 1
"3292"
127.0.0.1:6381> GET 1
"role-reported"
127.0.0.1:6381> GET 1
"master"
127.0.0.1:6381> GET 1
"role-reported-time"
127.0.0.1:6381> GET 1
"555550"
127.0.0.1:6381> GET 1
"config-epoch"
127.0.0.1:6381> GET 1
"0"
127.0.0.1:6381> GET 1
"num-slaves"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"num-other-sentinels"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"quorum"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"failover-timeout"
127.0.0.1:6381> GET 1
"6000"
127.0.0.1:6381> GET 1
"parallel-syncs"
127.0.0.1:6381> GET 1
"1"
127.0.0.1:6381>

SENTINEL 1
```

Slika 14: Demonstracija replikacije, konfigurisani Sentineli

U prvom koraku možemo da vidimo da replikacija funkcioniše – podataka unet na master instanci je prosleđen do obe replike. Takođe, u donjem desnom terminalu možemo da vidimo da je klaster dobro konfigurisan – Sentinel vidi druga dva Sentinelu, master i obe replike. Čitanje iz replika nije moguće, jer su u pitanju read-only replike, kao što je demonstrirano na slici iznad.

```
uros@uros-VirtualBox: ~/redis-lab
uros@uros-VirtualBox:~/redis-lab$ docker logs --tail 100 redis_master
1:X 09 Jul 2024 22:02:04.804 # +config-update-from sentinel 9b1d0db534f59896a8ee0dfa2197422b85d17dcd 172.18.0.7 5002 @ mymaster 172.18.0.2 6379
1:X 09 Jul 2024 22:02:04.804 # +switch-master mymaster 172.18.0.2 6379 172.18.0.4 6381
1:X 09 Jul 2024 22:02:04.805 * +slave slave 172.18.0.3:6380 172.18.0.3 6380 @ mymaster 172.18.0.4 6381
1:X 09 Jul 2024 22:02:04.807 * +slave slave 172.18.0.2:6379 172.18.0.2 6379 @ mymaster 172.18.0.4 6381
1:X 09 Jul 2024 22:02:04.813 * Sentinel new configuration saved on disk
1:X 09 Jul 2024 22:02:09.812 # +sdown slave 172.18.0.2:6379 172.18.0.2 6379 @ mymaster 172.18.0.4 6381
1:X 09 Jul 2024 22:02:28.183 # -sdown slave 172.18.0.2:6379 172.18.0.2 6379 @ mymaster 172.18.0.4 6381
1:X 09 Jul 2024 22:02:38.151 * +convert-to-slave slave 172.18.0.2:6379 172.18.0.2 6379 @ mymaster 172.18.0.4 6381

uros@uros-VirtualBox:~$ redis-cli -p 6379
127.0.0.1:6379> SET 2 "second"
(error) READONLY You can't write against a read only replica.
127.0.0.1:6379> GET 2
"SECOND"
127.0.0.1:6379>

REPLICA

uros@uros-VirtualBox:~$ redis-cli -p 6380
127.0.0.1:6380> SET 2 "SECOND"
(error) READONLY You can't write against a read only replica.
127.0.0.1:6380> GET 2
"SECOND"
127.0.0.1:6380>

REPLICA

uros@uros-VirtualBox:~$ redis-cli -p 6381
127.0.0.1:6381> SET 2 "SECOND"
OK
127.0.0.1:6381> GET 2
"SECOND"
127.0.0.1:6381>

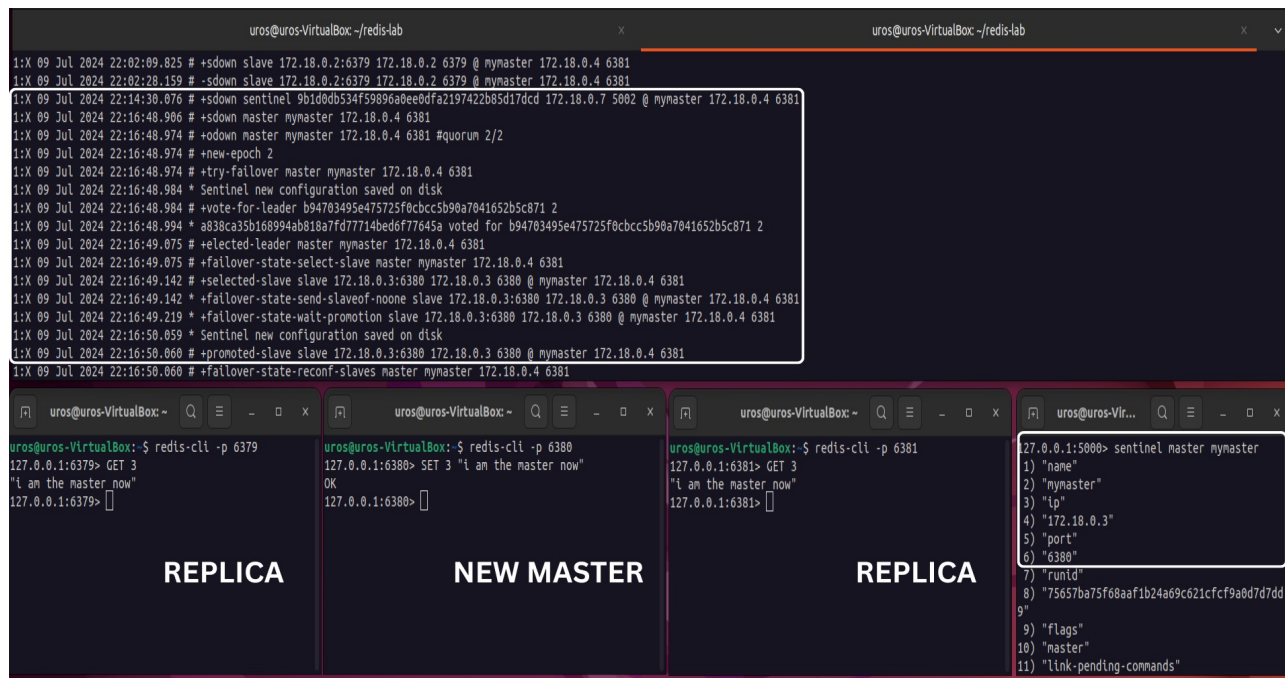
NEW MASTER

uros@uros-VirtualBox:~$ redis-cli -p 6381
127.0.0.1:6381> GET 1
"641"
127.0.0.1:6381> GET 1
"last-ping-reply"
127.0.0.1:6381> GET 1
"641"
127.0.0.1:6381> GET 1
"down-after-milliseconds"
127.0.0.1:6381> GET 1
"5000"
127.0.0.1:6381> GET 1
"info-refresh"
127.0.0.1:6381> GET 1
"3292"
127.0.0.1:6381> GET 1
"role-reported"
127.0.0.1:6381> GET 1
"master"
127.0.0.1:6381> GET 1
"role-reported-time"
127.0.0.1:6381> GET 1
"555550"
127.0.0.1:6381> GET 1
"config-epoch"
127.0.0.1:6381> GET 1
"0"
127.0.0.1:6381> GET 1
"num-slaves"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"num-other-sentinels"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"quorum"
127.0.0.1:6381> GET 1
"2"
127.0.0.1:6381> GET 1
"failover-timeout"
127.0.0.1:6381> GET 1
"6000"
127.0.0.1:6381> GET 1
"parallel-syncs"
127.0.0.1:6381> GET 1
"1"
127.0.0.1:6381>

SENTINEL 1
```

Slika 15: Automatski failover

Nakon prvog koraka je simuliran otkaz mastera *sleep* komandom (master je uspavan na 30 sekundi). U gornjem terminalu možemo da prvo da vidimo proces automatskog failover-a, gde *redis_slave_2:6381* instanca preuzima ulogu mastera, a zatim, nakon isteka 30 sekundi, stari master postaje ponovo dostupan i biva proglašen za repliku novog mastera. Promenu mastera možemo da potvrdimo na osnovu logova Sentinela (donji-desni terminal). Takođe, upisom podataka u novi master vidimo da je replikacija nastavlja neometano.



Slika 16: Otkaz jednog Sentinela

Na slici iznad je demonstrirana robusnost korišćenje arhitekture. Zahvaljujući činjenici da je svaka instanca na zasebnom čvoru, čak i otkazom jednog Sentinela i mastera, druga dva Sentinela mogu uspešno da odrade automatski failover, jer imaju većinu.

Ovakva arhitektura predstavlja samo jedno od mnogih drugih rešenja za postizanje visoke dostupnosti. U realnom sistemu je potrebno efikasno iskoristiti resurse koji su nam na raspolaganju, vodeći računa pritom da raspodelom instanci po čvorovima ne uvedemo jedinstvenu tačku otkaza u sistem.

3.2.3 – Pretplata na obaveštenja Sentinela

Jedna od mnogobrojnih mogućnosti Redis Sentinela jeste i obaveštavanje administratora sistema, ili drugih podstistema o događajima koji se dešavaju u klasteru. U te svrhe Sentinel koristi Redis PUB/SUB mehanizam i objavljuje (*eng. Publish*) sve događaje koji se dešavaju u sistemu na odgovarajućim kanalima. Korišćenjem Redis klijenta, iz drugih aplikacija je moguća pretplata (*eng. Subscribe*) na ove događaje, ukoliko je potrebno da se na njih reaguje. Neki od događaja koje Redis Sentinel prati su:

- **+slave** – nova replika je dodata u sistem
- **+sentinel** – novi Sentinel je dodat u sistem
- **+failover-end** – proces failover-a je završen uspešnost
- **+sdown** – neka instanca (Redis, Sentinel) je otkazala
- **-sdown** – instanca je ponovo dostupna
- **+try-failover** – pokušaj failover-a je iniciran od strane Sentinela
- **+elected-leader** – nova master replika je izglasana

Navedeni su samo neki od događaja koji mogu biti od interesa. Osim navedenih postoje i brojni drugi događaji o kojima Sentinel može da obavesti ostatak sistema[3]. Imena kanala po kojima se događaji emituju imaju isti naziv kao i sam događaj. Primer pretplate na kanal **+sdown**, kako bismo pratili otkaz komponenti u sistemu prikazan je na slici 17.

```

uros@uros-VirtualBox:... x uros@uros-VirtualBox:... x uros@uros-VirtualBox:...
127.0.0.1:5000> SUBSCRIBE +sdown
1) "subscribe"
2) "+sdown"
3) (integer) 1
1) "message"
2) "+sdown"
3) "master myprimary 127.0.0.1 6379"
1) "message"
2) "+sdown"
3) "slave 127.0.0.1:6379 127.0.0.1 6379 @ myprimary 127.0.0.1 6380"
Reading messages... (press Ctrl-C to quit or any key to type command)

```

Slika 17: Pretplata na **+sdown** kanal

3.2.4 – TILT mod

Redis Sentinel zavisi od sistemskog časovnika. Da bi utvrdio koje instance nisu više dostupne, Sentinel upoređuje vremena poslednjeg odgovora instanci na PING komandu i upoređuje je sa trenutnim vremenom. Ukoliko se vreme iznenada promeni iz bilo kog razloga, Sentinel može da počne čudno da se ponaša.

Zbog toga Sentinel sadrži i specijalni TILT mod, u koji prelazi ukoliko dođe do opisane situacije. U ovom modu, Sentinel nastavlja da nadgleda sistem, međutim prestaje da preduzima bilo kakve akcije (pokretanje failover-a, glasanje, detektovanje otkaza) i na komandu **SENTINEL is-master-down-by-addr** uvek odgovara odrično – jer se Sentinelu u TILT modu ne može verovati da pouzdano detektuje otkaze. Ukoliko nakon 30 sekundi vreme prestane drastično da se menja, Sentinel se vraća na normalni mod rada. Primer prelaska u TILT mod je prikazan na slici 18.

```

# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_tilt_since_seconds:-1
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=myprimary,status=ok,address=127.0.0.1:6380,slaves=1,sentinels=2
127.0.0.1:5000>

```

Normalan rad

```

35887:X 10 Jul 2024 01:20:56.166 * Sentinel ID is f32f5e8476a48829b5e7cf1de7c12902dd9e0c95
35887:X 10 Jul 2024 01:20:56.166 # +monitor master myprimary 127.0.0.1 6380 quorum 2
35887:X 10 Jul 2024 01:20:30.056 # +tilt #tilt mode entered
35887:X 10 Jul 2024 01:18:30.014 # +tilt #tilt mode entered

```

Logovi sentinela - Promena časovnika

```

# Sentinel
sentinel_masters:1
sentinel_tilt:1
sentinel_tilt_since_seconds:14
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=myprimary,status=ok,address=127.0.0.1:6380,slaves=1,sentinels=2
127.0.0.1:5000>

```

TILT

Slika 18: TILT mod Redis Sentinel

Na slici je komandom *info* možemo da dobijemo informaciju o tome u kom se modu Sentinel nalazi. Kada je u normalnom modu možemo da vidimo da je indikator *sentinel_tilt* postavljen na 0, dok je u tilt modu njegova vrednost 1. Takođe, u TILT modu možemo da vidimo koliko se dugo Sentinel nalazi u tom stanju.

Da bi detektovao vremenske anomalije, Sentinel oko 10 puta u sekundi (otprilike svakih 100 milisekundi) proverava *timestamp* sistema. Samim tim on očekuje da vremenska razlika između dva *timestamp*-a iznosi približno 100 milisekundi. Ukoliko se ova vrednost drastično razlikuje, prelazi se u TILT mod.

Pored svih navedenih tehnika, koje se odnose na replikaciju i automatski failover – koje predstavljaju suštinsku osnovu visoko dostupnih sistema, postoje još neke tehnike koje se paralelno koriste za postizanje boljih performansi sistema koji koriste Redis. Redis klaster – za skaliranje Redis baze na više čvorova particionisanjem podataka, čime se postižu značajne performanse čitanja, zatim geografskom distribucijom čvorova i Active-Active replikacijom se obezbeđuju bolje performanse operacija upisa, po ceni slabije konzistentnosti, ali i neometano funkcionisanje dela sistema ukoliko neki drugi deo otkáže (na primer u slučaju geografskih nepogoda). Ove tehnike nisu obrađene, jer prevazilaze temu ovog seminarskog rada, u kome je akcenat na visokoj dostupnosti, a ne toliko na skaliranju.

ZAKLJUČAK

Visoka dostupnost je ključni aspekt savremenih baza podataka, posebno u okruženjima gde je kontinuirani pristup podacima od suštinskog značaja. U samom radu je dato obrazloženje odakle potreba za ovakvim sistemima i koje su posledice čak i minimalnog narušavanja dostupnosti. Redis, kao visoko performantna baza podataka, nudi robusna rešenja za postizanje visoke dostupnosti koristeći mehanizme replikacije i automatskog failover-a korišćenjem Redis Sentinela.

Pored gotovih rešenja, potrebno je primenjivati i tehnike kao što su inženjering haosa i redovno testiranje sistema u produkciji, koje pomažu u identifikaciji slabosti i potencijalnih problema pre nego što se oni ozbiljnije manifestuju.

LITERATURA

- [1] Running Redis at scale, Redis - <https://redis.io/learn/operate/redis-at-scale>
- [2] Redis Replication, Redis - https://redis.io/docs/latest/operate/oss_and_stack/management/replication/
- [3] Redis High-Availability with Redis Sentinel, Redis - https://redis.io/docs/latest/operate/oss_and_stack/management/sentinel/
- [3] The ultimate guide to database High-Availability, Percona - <https://www.percona.com/blog/the-ultimate-guide-to-database-high-availability/>
- [4] Patterns to achieve High-Availability, Canonical - <https://ubuntu.com/blog/database-high-availability>
- [5] A brief history of High-Availability, Cockroachlabs - <https://www.cockroachlabs.com/blog/brief-history-high-availability/>
- [6] NoSQL Databases, Wikipedia - <https://en.wikipedia.org/wiki/NoSQL>
- [7] CAP theorem, Wikipedia - https://en.wikipedia.org/wiki/CAP_theorem
- [8] Cost of datacenter outages, Ponemon institute - https://www.vertiv.com/globalassets/documents/reports/2016-cost-of-data-center-outages-11-11_51190_1.pdf