**Univerzitet u Nišu, Elektronski fakultet**

logo_1960_4**Katedra za računarstvo**

*High availability rešenja kod PostgreSQL-a*

**Stefan Živić 698**

**Predmet: Sistemi za upravljanje bazama podataka**

Sadržaj

[Uvod 3](#_Toc75710907)

[HA 4](#_Toc75710908)

[Kako funkcioniše HA? 4](#_Toc75710909)

[Merenje dostupnosti 5](#_Toc75710910)

[Uzroci nedostupnosti sistema (downtime-a) 6](#_Toc75710911)

[Trajnost i dostupnost 8](#_Toc75710912)

[HA pomoću Patroni-a 9](#_Toc75710913)

[Zašto koristimo HAProxy? 10](#_Toc75710914)

[Zašto koristimo etcd? 11](#_Toc75710915)

[Zašto koristimo Patroni? 11](#_Toc75710916)

[Patroni softverski stek 12](#_Toc75710917)

[Konfigurisanje etcd sloja 13](#_Toc75710918)

[Konfigurisanje Patroni-a 16](#_Toc75710919)

[Konfigurisanje HAProxy komponente 22](#_Toc75710920)

[Testiranje HA rešenja – primeri 24](#_Toc75710921)

[Promena master čvora(Performing a managed switchover) 24](#_Toc75710922)

[Prekid usluga servera (outage) 26](#_Toc75710923)

[Vraćanje instance u klaster 26](#_Toc75710924)

[Dodavanje instanci u klaster 28](#_Toc75710925)

[Zaključak 30](#_Toc75710926)

[Literatura 31](#_Toc75710927)

# Uvod

High availability (u daljem tesktu HA) nije samo industrijski pojam. Činjenica da nešto treba biti dostupno gotovo uvek, obično znači i da je izuzetno važno i da usluga mora biti dostupan klijentima, bez obzira na cenu. Baza podataka je centralni deo gotovo svakog sistema pa je zbog toga dostupnost usluge ključna. Ako nema baze - nema ni ostatka sistema.

Uzećemo za primer sisteme bazirane na cloud-u (oblaku). Zahvaljujući njima, našim aplikacijama, platformama, uslugama, pa čak i medijima je lakši pristup nego ikad. Sve dok imate internetsku vezu, možete se prijaviti u sistem koji čuva vaš rad, pamti gde ste stali sa gledanjem filma, koji proizvod ste prethodno pregledali itd. Ipak, šta se dešava kada dođe do zastoja u isporučenju takvih usluga? Ispostavilo se da nas to može koštati prilično novca. Zbog toga je toliko važno pronaći uslugu na koju možete da se oslonite, onu koja može da garantuje neprekidnu upotrebu i obećava „visoku dostupnost“ - ali tu postaje nezgodno.

Što se tiče organizacije rada, na početku je objašnjeno šta je HA, kako funkcioniše i kako se meri. Dat je i pregled planiranih i neplaniranih uzroka za nedostupnost sistema. Objašnjena je razlika između trajnosti i dostupnosti, obzirom da ta dva pojma često idu zajedno. Centralni deo rada, a i sam njegov praktični deo, predstavlja kreiranje HA rešenja pomoću Patroni softvera. Instance servera podizane su na Linux Ubuntu operativnom sistemu, na verziji 16.04. Biće opisan svaki korak detaljno, šta je potrebno uraditi i na koji način. Takođe, za svaki od njiih videćemo koje je softvere potrebno instalirati i kako ih konfigurisati.

# HA

Kada je reč o zagarantovanom trajanju usluge, ono o čemu zaista razgovaramo je dostupnost te usluge. Visoka dostupnost odnosi se na dostupnost sistema, rešenja ili aplikacije koja može da funkcioniše bez ikakvih prekida tokom dužeg vremenskog perioda.

HA sistemi se koriste u situacijama i industrijama gde je presudno da sistem i dalje radi kad dođe do greške, kvara itd. Realni sistemi visoke dostupnosti uključuju vojnu kontrolu, autonomna vozila, industrijske i zdravstvene sisteme. Život ljudi zavisi od toga da li su ovi sistemi dostupni i da li funkcionišu u svakom trenutku. Na primer, ako sistem koji upravlja autonomnim vozilom ne funkcioniše dok vozilo radi, to može prouzrokovati nesreću, ugrožavajući putnike, druge vozače i vozila, pešake.

HA sistemi moraju biti dobro dizajnirani i temeljno testirani pre nego što se koriste. Planiranje i projektovanje jednog od ovih sistema podrazumeva da sve komponente ispunjavaju željeni standard dostupnosti. Mogućnosti sigurnosne kopije podataka i preusmeravanja imaju važnu ulogu u osiguravanju da HA sistemi ispunjavaju svoje ciljeve dostupnosti. Dizajneri sistema takođe moraju pažljivo obratiti pažnju na tehnologiju skladištenja podataka i pristupa koju koriste.

## Kako funkcioniše HA?

Gotovo je nemoguće da sistemi budu dostupni u 100% slučajeva, odnosno vremena, pa istinski sistemi visoke dostupnosti uglavnom teže standardu pet devetki operativnih performansi (biće objašnjeno u delu o [merenju dostupnosti](#_Merenje_dostupnosti)).

Sledeća tri principa koriste se pri projektovanju HA sistema kako bi se osigurala visoka dostupnost:

* **Pojedinačne tačke kvara- Single points of failure**: Jedna tačka kvara je komponenta koja bi prouzrokovala pad celog sistema ako dođe do greške u njenom radu. Ako se sistem zasniva na jednom serveru koji pokreće aplikaciju, taj server je potencijalna tačka neuspeha. Ako taj server zakaže, aplikacija neće biti dostupna.
* **Pouzdano preuzimanje (prelaz) - Reliable crossover**  : Bitno je imati više kopija komponenti. U tom slučaju, kada dođe do greške u jednoj od komponenti, neophodno je osigurati pouzdano  preusmeravanje, što je čin prelaska sa komponente X na komponentu Y bez gubitka podataka ili uticaja na performanse.
* **Otkrivanje kvarova - Failure detectability:**  Kvarovi moraju biti vidljivi, a idealno je da sistemi imaju ugrađenu automatizaciju da se sami nose sa kvarovima. Takođe bi trebalo da postoje ugrađeni mehanizmi za izbegavanje kvara uobičajenog uzroka, gde dva ili više sistema ili komponenti istovremeno otkazuju, verovatno iz istog uzroka.

Da bi se osigurala visoka dostupnost kada mnogo korisnika istovremeno pristupi sistemu, **balans opterećenja (load balancing)**  postaje neophodan. Balansiranje opterećenja automatski raspoređuje zahteve na sistemske resurse. Odlučuje se koji sistemski resurs je najsposobniji za efikasno rukovanje nekim zahtevom. Korišćenje više komponenti za raspodelu osigurava to da nijedan resurs nije preopterećen.

Serveri u HA sistemu su u klasterima i organizovani su u slojevitu arhitekturu da odgovore na zahteve balansa opterećenja. Ako jedan server u klasteru ne uspe, replicirani server u drugom klasteru može se nositi s zadatkom koji je dodeljen serveru koji iz nekog razloga nije mogao da ga obavi. Ova vrsta redudantosti omogućava da uz minimalan uticaj na performanse sekundarna komponenta preuzima posao primarne komponente kada ona otkaž.

Što je sistem složeniji, teže je osigurati visoku dostupnost, jer u složenom sistemu jednostavno postoji više tački kvara.

## Merenje dostupnosti

Dostupnost se, u industriji koja se bavi softverom, uglavnom meri korišćenjem dve vrste metrika (merenja):

* *Srednje vreme za oporavak - The mean time to recover (****MTTR****)*
* *Srednje vreme između kvarova - The mean time between failures (****MTBF****)*

Za većinu scenarija u kojima dolazi do greške, a samim tim i nedostupnosti, sistema - *downtime*, industrija se usredsređuje na probleme vezane za srednje vreme za oporavak (MTTR) i istražuje kako optimalno dizajnirati sisteme da ih smanje. MTBF je generalno više primenljiv na pokazatelje dostupnosti hardvera. Međutim, s obzirom na to da možete dizajnirati stvarne aplikativne klastere kako biste izbegli pojedinačne tačke kvara (razloge prekida), kvarovi komponenata ne moraju nužno rezultirati nedostupnošću aplikacije. Stoga stvarni klasteri aplikacija mogu u velikoj meri smanjiti MTBF sa stanovišta dostupnosti aplikacija.

Druga metrika koja se obično koristi je **pravilo broja devetki**. Na primer, 526 minuta nedostupnosti sistema svake godine rezultira 99,9% ili tri devetke. Pet minuta nedostupnosti sistema za svaku godinu rezultira 99,999% ili pet devet dostupnosti. Naizgled, razlika između 99,99% i 99,9% dostupnosti možda ne izgleda previše dramatično. Ali kada detaljno analizirate šta ove brojke znače, prava priča počinje da se pokazuje.

Koristeći jednostavan kalkulator vremena rada, možemo bolje razumeti šta ova merenja znače:

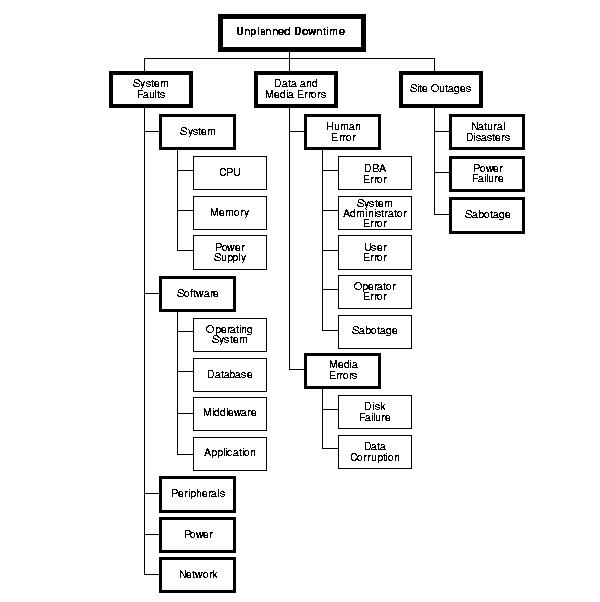
|  |  |  |  |
| --- | --- | --- | --- |
| DOSTUPNOST | 99% | 99,9% | 99,99% |
| Svakodnevno | 14m 24s | 1m 26.4s | 8.6s |
| Nedeljno | 1hr 40m 48s | 10m 4.8s | 1m 0.5s |
| Mesečno | 7hr 18m 17.5s | 43m 49.7s | 4m 23s |
| Godina | 3d 15hr 39m 29,5s | 8h 45m 57s | 52m 35,7s |

Kao što vidite, razlika između 99,99% i 99,9% dostupnosti znači dodatnih 7 sati zastoja za tu uslugu svuda. To dalje znači 1 minut i 26 sekundi svakodnevnog zastoja ili 10 minuta i 48 sekundi svake nedelje.

*Tabela 1 - Merenje dostupnosti*

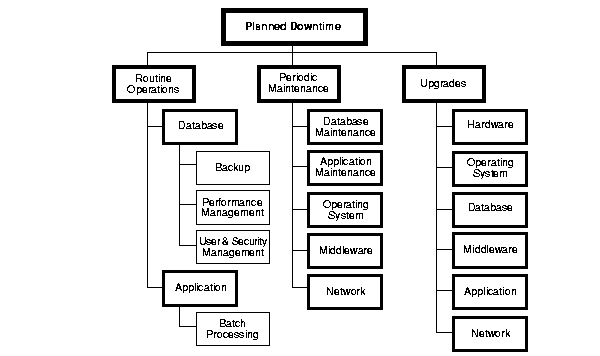
## Uzroci nedostupnosti sistema (downtime-a)

Zastoji i nedostupnost mogu biti *planirani ili neplanirani*. Slika 1 prikazuje mnoge uzroke neplaniranog zastoja. Upravo zbog toga što su neplanirani mogu biti jako nezgodni za poslovanje, jer se ne može predvideti kada će se desiti i koliko će trajati.



Slika 1 - Neplanirani uzroci nedostupnosti sistema

Sa druge strane, planirani zastoji mogu biti podjednako nezgodni za poslovanje, posebno u globalnim preduzećima koja podržavaju korisnike u više vremenskih zona. Slika 2 prikazuje nekoliko vrsta planiranog zastoja. Generalno se mogu klasifikovati kao rutinske operacije, periodično održavanje i nadogradnje.



Slika 2 - Planirani uzroci nedostupnosti sistema

## Trajnost i dostupnost

Pri dizajniranju sistema za skladištenje podataka važno je razlikovati svojstva dostupnosti i trajnosti. *Dostupnost* je mera verovatnoće da je sistem sposoban da prihvati čitanje i pisanje u bilo kom trenutku u vremenu. Trajnost je mera verovatnoće da kada je jednom upis bio prihvaćen (u PostgreSQL-u ovo znači uspešno izvršenje transakcije), te informacije se neće izgubiti. Svi stvarni sistemi moraju da naprave kompromis između dostupnost, trajnosti, performansi i cene.

Za jednostavan primer možemo razmotriti klaster koji se sastoji od dva servera baze podataka. Jedan je pokrenut kao primarni server i prihvata upise iz naše aplikacije. Drugi je u stanju pripravnosti, replicira promene sa primarnog servera, da bi ako se nešto loše dogodi, poput pada, mogao da postane primarni. Pitanje na koje sada moramo da odgovorimo je koliko promena možemo priuštiti da izgubimo i koliko često. U nekim sistemima, poput onih za prikupljanje statistike senzora, nije velika tragedija ako dođe do zastoja od nekoliko minuta. U drugim slučajevima, poput finansijskih sistema, bićemo u velikim problemima ako čak i jedna transakcija nestane Ako je gubljenje nekih podataka u redu, tada primarni sistem može prijaviti uspeh čim uspešno obradi podatke. To se naziva asinhrona replikacija. Ako primarni sever zakaže u ovom sistemu, postoji prilično velika šansa da se nekoliko transakcija neće replicirati i zbog toga će biti izgubljene.

S druge strane, ako treba da osiguramo da se podaci ni u kom slučaju ne izgube, želeli bismo da sačekamo potvrdu da je rezervni server uspešno upisao podatke na disk. Međutim, šta se dešava ako on ne uspe? Ako gubitak podatka u bilo kom pojedinačnom čvoru u okviru klastera nije prihvatljiv, tada naš sistem mora da prestane da prihvata transakcije i da javi grešku. Sistem će ostati nedostupan dok se ne uspostavi novi čvor za repliciranje. Ako su i visoka dostupnost i trajnost potrebni, tada su neophodna najmanje tri čvora baze podataka.

Treba napraviti mnogo kompromisa. PostgreSQL sistem replikacije čak omogućava postavljanje replikacionog klastera gde je garanciju trajnosti obezbeđuje se strimingom logova transkacija u namenski sistem za skladištenje i dostupnost obezbeđuje pomoću tzv. *Hot standby* principa.

# HA pomoću Patroni-a

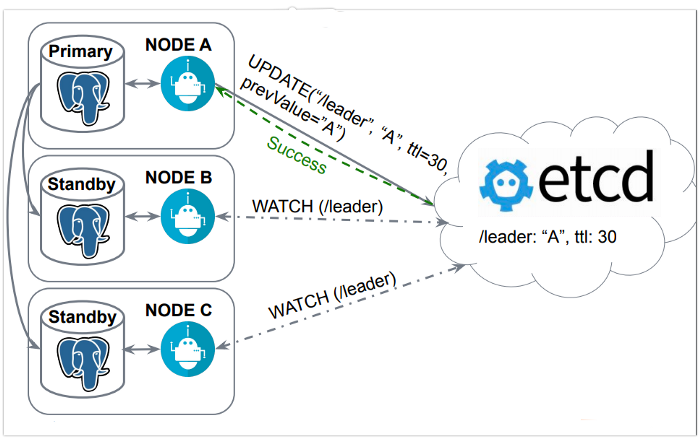
Patroni je softver za upravljanje HA koji je razvio Zalando za orkestriranje i automatizaciju nekoliko aspekata PostgreSQL klastera. Radi pomoću niza posebnih komponenti sa kojima komunicira kao sa slojevima u steku, kako bi na kraju proizveo softver koji je dostupan u visokoj meri.

Razlog zbog kojeg Patroni radi kao niz softverskih slojeva je smanjenje zavisnosti od bilo koje tačke otkaza - *single point of failure*. Uvek postoje element za upravljanje konsenzusom i resurs za udruživanje. Svaki deo steka dizajniran je za rad na svim serverima kandidatima. Zbog toga više ne moramo da se oslanjamo na virtuelnu IP adresu ili Canonical Name Record (CNAME) koji mora biti dodeljen promovisanoj replici.

Na kraju, Patroni bi trebalo da isporuči klaster sa sledećim mogućnostima:

* Može automatski izabrati zamenu u slučaju otkaza
* Može preusmeriti veze sposobne za pisanje na novoizabrani primarni čvor
* Novi čvorovi mogu se dodati u klaster
* Oporavljeni primarni čvorovi mogu se ponovo pridružiti klasteru kao replike

Rešenje koje će biti opisano, pored centrale komponente Patroni, koristi i HAProxy i etcd. Svako od njih ima svoju ulogu u sistemu i razlog zbog koga je baš on izabran, pa da vidimo koji su to razlozi.



Slika 3 - Arhitektura sistema

### Zašto koristimo HAProxy?

Da bismo poboljšali naše rešenje za softver koji je dostupan u visokoj meri, potrebno je da razumemo njegove slabosti. Neka druga rešenja, poput onog koje se zasniva na korišćenju *repmgr*-a, zahtevaju upotrebu virtuelne IP adrese. Glavni problem u vezi sa tim je što se virtuelni IP može deklarisati sa bilo kog mesta i, ako se ne upravlja pravilno, može dovesti do *split brain* scenaria.

U ostalim slučajevima, u slučaju otkaza, potrebna nam je virtuelna IP adresa da bi aplikacija nastavila da pristupa bazi podataka. Da bi smanjio ovu zavisnost, HAProxy će uvek “razgovarati” sa glavnim čvorom. U HAPrpxy-ju  moramo dati port za proveru stanja ispravnosti instance. 8008 je podrazumevani port za Patroni REST API. HA proksi će poslati GET zahtev svim čvorovima u klasteru u portu 8008. Samo glavni čvor će dati status 200 OK. Ostali čvorovi će vratiti 503 Server nedostupan.

HAProxy nema to ograničenje, jer se svaka IP adresa ponaša kao da je primarni čvor. Dakle, sve dok se povezujemo preko proxy porta koji odaberemo tokom konfiguracije, komuniciramo sa bilo kojim čvorom koji je primarni u tom određenom trenutku. HAProxy je zapravo sloj koji se bavi rutiranjem kod našeg rešenja.

### Zašto koristimo etcd?

Da bismo izgradili stek, biće nam potreban pouzdan sloj za prosleđivanje poruka. Studenti sa Univerziteta Stanford smislili su konsenzusni algoritam koji su nazvali Raft. Postoji mnogo teorija o tome kako to funkcioniše, ali se svodi nato da par ključ / vrednost uskladišten u sloju zasnovanom na Raft-u, ostaje dosledan na svim serverima i generalno je otporan na deljenja mreže (*network partitions)*  i *split-brain* scenario. Za razliku od repmgr-a, koji algoritam implementira direktno, Patroni ovo čini modularnom komponentom.

Podrazumevani softver koji Patroni koristi za Raft je **etcd**. Ovo je od presudnog značaja jer ćemo koristiti etcd za skladištenje lokacije primarnog Postgres servera. Pod uslovom da imamo servis koji se može povezati sa etcd, bilo koji od naših Postgres servera odmah će znati lokaciju primarnog sistema. Zbog toga je trivijalno menjati izvori replikacije kada se primarni sistem promeni.

### Zašto koristimo Patroni?

Patroni predstavlja centralni deo sistema i povezuje komponente o kojima je bilo reči do sada. Deluje kao glavni koordinator i ima brojne uloge. Ovo je proces koji se ponavlja na svakom PostgreSQL serveru:

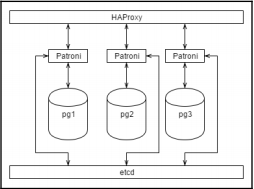
1. Proverava prisustvo postojećeg primarnog servera u sloju Raft.
2. Ako nije pronađen primarni server, on ubacuje ključ u sloj Raft i postaje primarni čvor.
3. Ako je ovaj trenutni čvor primarni, signalizira se sloju HAProxy da ga koristi kao novi cilj za preusmeravanje
4. Ako se pronađe primarni server, on vrši nekoliko provera i pokušava da transformiše trenutni server u repliku (jednu od replika u klasteru).

Patroni ponavlja ove korake svakih nekoliko sekundi na svakom serveru na kome je instaliran. Kao posledica, neki prekidi mogu rezultirati *race condition-*om u kojima će višestruke replike pokušati da postanu novi primarni server. Raft sloj, ili u našem slučaju etcd, osigurava da će samo jedan pobediti u ovoj trci, a Patroni se brine za ostalo.

Ovaj način rada takođe omogućava svakom Postgres serveru da radi nezavisno, tako da ne postoji nijedna pojedinačna tačka kvara - *single point of failure*. Obzirom da se replike paralelno preusmeravaju na novi primarni server, ceo klaster ima svojstvo “samoizlečenja” , tj postaje kako se to kaže *self-healing swarm.*

## Patroni softverski stek

Arhitekturni diagram sistema koji pravimo izgleda kao na sledećoj slici :



Slika 4 - Softverski stek Patroni rešanja

Vidimo da svaki Patroni element komunicira samo sa svojom lokalnom instancom Postgres servisa. Takođe komunicira sa etcd-om i HAProxy-em kako bi klaster održao u konzistentnom i “zdravom” stanju. Budući da svaki od ovih vertikalnih elemenata radi nezavisno, mogli bismo da nastavimo da dodajemo Postgres čvorove kojima upravlja Patroni gotovo neograničeno. Ali,  prvo moramo da izgradimo ovakav sistem.

Sada ćemo videti kako da pokrenemo sve potrebne servise. Patroni, etcd i HAProxy imaju brojne zavisnosti koje su im potrebne da bi funkcionisale. Većina njih se lako može instalirati i tu ne bi trebalo da bude većih problema. Kao što se može videti i sa diagrama potrebno nam je bar 3 Postgres servera, odnosno čvora. Za svrhe dalje demonstracije nazvaćemo ih **node1, node2 i node3**.

|  |  |  |
| --- | --- | --- |
| **HOSTNAME** | **IP ADDRESS** | **PURPOSE** |
| Node1 | 192.168.0.109 | PostgreSQL, Patroni |
| Node2 | 192.168.0.107 | PostgreSQL, Patroni |
| Node3 | 192.168.0.110 | PostgreSQL, Patroni |
| Node1 | 192.168.0.109 | etcd node 1 |
| Node2 | 192.168.0.107 | etcd node 2 |
| Node3 | 192.168.0.110 | etcd node 3 |
| Node4 | 192.168.0. | HAProxy |

Kao što je rečeno u uvodu, instance servera podignute su na Ubuntu 16.04. Na svakom od njih potrebno je uraditi sledeće kako bismo instalirali neophodne zavisnosti:

*Tabela 2 – Pregled instanci i njihovih uloga*

*sudo apt-get install python3-psycopg2 python3-pip python-yaml*

Prvi na listi je Pythonov *psycopg2* interfejs zaPostgres. Patroni ovo koristi za povezivanje sa Postgresom za razne operacije. Dalje instaliramo *pip*, Python-ov alat za instaliranje paketa iz *Python Package Index (PyPI)*. Ovo je vrlo slično PostgreSQL Extension Network alatu, ali za popularne Python pakete. *YAML* je skraćenica za *Yet Another Markup Language*. To je format koji neki projekti koriste za definisanje konfiguracionih fajlova. Među njima se nalazi i Patroni. Python API koji komunicira sa ovim datotekama zapravo se zove *PyYAML*, ali sistemi zasnovani na Debianu su ga preimenovali u *python-yaml* kako bi odgovarao odabranoj šemi imenovanja.

## Konfigurisanje etcd sloja

Da bi Patroni pouzdano utvrdio ili definisao identitet primarne instance PostgreSQL, potreban nam je distribuirani sloj u kome ćemo čuvati ključ/vrednost parove. Kao što je rečeno, za tu ulogu koristićemo etcd.

Na svakom od 3 čvora sa potrebno je uraditi sledeće:

1. Raspakovati arhivu sa etcd-om i “instalirati” ga kao korisnik sa *root* privilegijama*.* U mom slučaju to je izgledalo ovako:

*tar -xf etcd-v3.3.18-linux-amd64.tar.gz*

*sudo cp etcd-v3.3.18-linux-amd64/etcd\* /usr/local/bin*

1. Kreirati direktorijum za podatke (storage) za etcd, ponovo kao korisnik sa root privilegijama i postaviti podrazumevanog postgres korisnika kao vlasnika:

*sudo mkdir /db/etcd*

*sudo chown postgres:postgres /db/etcd*

1. Kreirati fajl /etc/etcd.conf na *node1* instanci koji će izgledati ovako:

*name: node1*

*data-dir: /db/etcd*

*initial-advertise-peer-urls: http://node1\_ip:2380*

*listen-peer-urls: http://0.0.0.0:2380*

*listen-client-urls: http://0.0.0.0:2379*

*advertise-client-urls: http://node1\_ip:2379*

*initial-cluster: "node1=http://node1\_ip:2380,node2=http://node2\_ip:2380,node3=http:// node3\_ip:2380"*

1. Kreirati fajl /etc/etcd.conf na *node2* instanci koji će izgledati ovako:

*name: node2*

*data-dir: /db/etcd*

*initial-advertise-peer-urls: http://node2\_ip:2380*

*listen-peer-urls: http://0.0.0.0:2380*

*listen-client-urls: http://0.0.0.0:2379*

*advertise-client-urls: http://node2\_ip:2379*

*initial-cluster: "node1=http://node1\_ip:2380,node2=http://node2\_ip:2380,node3=http:// node3\_ip:2380"*

1. Kreirati fajl /etc/etcd.conf na *node3* instanci koji će izgledati ovako:

*name: node3*

*data-dir: /db/etcd*

*initial-advertise-peer-urls: http://node3\_ip:2380*

*listen-peer-urls: http://0.0.0.0:2380*

*listen-client-urls: http://0.0.0.0:2379*

*advertise-client-urls: http://node3\_ip:2379*

*initial-cluster: "node1=http://node1\_ip:2380,node2=http://node2\_ip:2380,node3=http:// node3\_ip:2380"*

1. Startovati etcd daemon pomoću komande:

*etcd --config-file /etc/etcd.conf*

Počinjemo sa preuzimanjem i instaliranjem etcd-a tako da imamo distribuirani sloj komunikacije koji Patroni koristi. Datoteka koju preuzmemo takođe treba da sadrži dokumentaciju, ali nama je potrebno samo da instaliramo etcd i etcdctl. Slično kao i PostgreSQL, etcd takođe koristi **write-ahead log (WAL)** za trajnost podataka. Zbog toga nam je potrebno mesto za skladištenje ovih WAL podataka. Podrazumevano, etcd će stvoriti poddirektorijum od mesta na kojem je pokrenut, što zapravo ne želimo ako nam je namera da svaki put uspostavimo i komuniciramo sa istim klasterom etcd.

Sada moramo konfigurisati etcd na svim PostgreSQL serverima koji čine klaster Patroni. Počinjemo tako što čvor imenujemo parametrom *name*, a zatim definišemo WAL direktorijum o kojem smo ranije razgovarali parametrom ***data-dir*.** Usluga etcd održava *peer-to-peer* mrežu za čvorove koji međusobno komuniciraju. Podrazumevano, ova mreža radi na portu *2380* na svakom čvoru na kojem je pokrenut etcd, ali želimo izričito navesti ime hosta kako bismo bili sigurni da možemo da prihvatimo spoljašnje veze. Početna postavka **advertise-peer-urls** definiše ime i port koje bi drugi etcd čvorovi trebali koristiti prilikom komunikacije sa ovim sistemom.

Slično tome, parametar **listen-peer-urls** definiše koji će host i port nadgledati veze. U suštini, u ovom slučaju smo vezani za lokalne resurse i, prema dokumentaciji, ovde nije dobro navesti ime hosta. Dakle, koristili smo 0.0.0.0 da bismo se povezali sa bilo kojom adresom koju

koristi instanca na kojoj je pokrenut etcd. Ovo bi trebalo biti u redu dok nijedan drugi servis ne koristi port 2380.

Pored interne komunikacije, klijenti se obično povezuju sa etcd na portu 2379 radi skladištenja i preuzimanja key/value parova. Postavljanjem parametra **listen-client-urls** da sluša na 0.0.0.0, osigurali smo da Patroni može lokalno postavljati vrednosti, a bilo koji čvor u klasteru takođe može komunicirati sa etcd u slučaju da njihova lokalna etcd usluga nije dostupna . Slično oglašavanju sa drugima, svaki čvor se oglašava sa vrednošću u parametra **advertise-client-urls**, pa tako ime koristimo za spoljnu komunikaciju.

Konačno, možemo pokrenuti samu servis etcd i preusmeriti njegov izlaz u datoteku evidencije. Inače, etcd radi eksplicitno putem flag-ova ili promenljivih okruženja, ali to je pomalo nezgodno u poređenju sa stabilnošću konfiguracione datoteke. Stoga smo flag *--config-file* postavili na našu /*etc/etcd.conf* datoteku da bismo sprečili takvo ponašanje.

Većina Linux distribucija kao podrazumevani način za kontrolu servisa podrazumeva kontrolisanje pomoću *systemd.* Ako etcd želimo da kontrolišemo na taj način potrebno je da kreiramo *etcd.service* na putanji */etc/systemd/system,* koji izgleda ovako:

[Unit]

Description=etcd key-value store

Documentation=https://github.com/etcd-io/etcd

After=network.target

[Service]

User=postgres

Type=notify

ExecStart=/usr/local/bin/etcd --config-file /etc/etcd.conf

Restart=always

RestartSec=10s

LimitNOFILE=40000

[Install]

WantedBy=multi-user.target

Tada bismo mogli da aktiviramo, pokrenemo i zaustavimo etcd pomoću sledećih sistemctl naredbi:

***sudo systemctl enable etcd***

***sudo systemctl start etcd***

***sudo systemctl stop etcd***

Pored toga, naš izlaz iz log-a bio bi dostupan pomoću komande *journalctl*:

***journalctl -u etcd***

,što je mnogo lakše od procesa pisanja shell skripte za upravljanje servisima.

## Konfigurisanje Patroni-a

Patroni je primarna koordinaciona komponenta našeg steka. Kao što smo videli sa dijagrama na slici 4 u organizaciji steka, on je u određenoj mjeri uključen u svaki element steka. Iako povezuje sve elemente zajedno, njega ćemo obraditi sledeće zboh toga koliko je čvrsto povezan sa slojem *etcd* in *PostgreSQL-om.*

Ako je PostgreSQL server već pokrenut, Patroni će ga “prilagoditi” i iskoristiti. Ako ne, Patroni će kreirati novu instancu na osnovu načina na koji je konfigurisan. Već smo ustanovili da skladište distribuira iste informacije po celom klasteru, tako da prvi uspostavljeni server takođe postaje primarni čvor klastera. Bilo koja sledeća Instanca Patroni-a će započeti kao replika ili će se transformisati u nju.

Što se tiče instaliranja, najnovija verzija Patroni softvera bi trebalo da bude dostupna pomoću *pip* alata. Za instaliranje ostalih zavisnosti, najlakši način, mada ne jedini, je da se skine fajl *requirements.txt* koji je deo release-a za Patroni koji se mogu naći na: <https://github.com/zalando/patroni/releases>.

Kao što je više puta rečeno, imamo 3 čvora node1, node2 i node3. Na svakom od njih potrebno je uraditi sledeće:

* 1. Pomoću pip instalirati *patroni* softver i zavisnosti

**sudo pip3 install -r /path/to/requirements.txt**

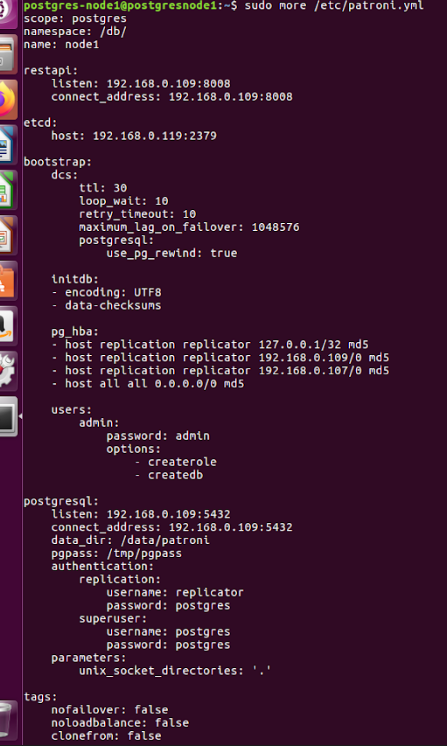
**sudo pip3 install patroni**

* 1. Kreirati direktorijum za konfigurisanje Patroni-a, i promeniti da vlasnik bude *Postgres* korisnik:

**sudo mkdir /etc/patroni**

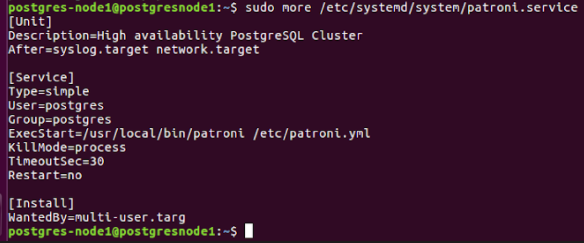
**sudo chown postgres:postgres /etc/patroni**

* 1. Kao *Postgres* korisnik kreirati file *patroni.yml* na putanji */etc/patroni* koji izgleda kao u primeru koji sledi. Sva pojavljivanja *node1* zameniti odgovarajućim imenom trenutne instance, kao i ip adrese node1 za odgovarajuće za node2 ili node3:



Slika 5 - Fajl Patroni.yml

* 1. Nakon toga kreiraćemo fajl za kontrolisanje Patroni servisa na putanji */etc/systemd/system/patroni.service*, koji izgleda ovako:

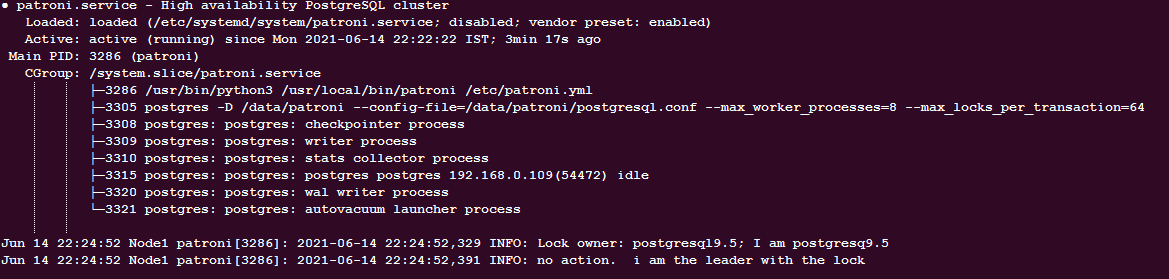


Slika 6 - Fajl Patroni.service

5. Pokrećemo servis komandom:

systemctl start patroni

Sada možemo da proverimo status servisa komandom systemctl status patroni i ako je sve kako treba on će izgledati slično kao na primeru:



Slika 7 - Status Patroni instance

Objasnićemo neke od prethodno pomenutih koraka. Što se tiče konfiguracionog fajla za Patroni, ova datoteka će definisati kako se inicijalizuje nova definicija klastera, trenutne parametre i postojeću strukturu *patronictl* alata za komandnu liniju. Počinjemo sa kreiranjem fajla nazvanog *patroni.yml* u direktorijumu /etc/patroni i osiguravamo da je on u vlasništvu korisnika *postgres*. To nam omogućava da potencijalno dodamo informacije o lozinki i osiguramo da ostanu poverljive i bezbedne u našem klasteru. Naziv čvora treba da se odnosi na ime servera da bi stvari bile jednostavne, ali ovo nije uslov.

U odeljku **restapi** definišemo dva parametra. Podesili smo *listen* na node1:8008, tako da Patroni posmatra port 8008 na imenovanom čvoru. Ovaj URL se može koristiti za dobijanje ili definisanje informacija o konfiguraciji ili za određivanje trenutnog primarnog servera. Postavili smo *connect\_address* na istu vrednost kako bi Patroni mogao da pristupi sopstvenom REST API-ju ako je neophodno. Ovi parametri se razlikuju u nekim posebnim slučajevima, ali u većini slučajeva to nije potrebno.

Posle toga je odeljak ***etcd***. Ovde definišemo lokaciju našeg distribuiranog sloja za čuvanje key-value parova. Zbog svoje jednostavnosti, potrebno je samo da postavimo parametar hosta na pgha1: 2379, isti klijentski interfejs i port koji smo definisali za etcd. Dalje, definišemo **bootstrap** odeljak i on sadrži nekoliko podelemenata. Svi parametri koje definišemo u okviru ovih pododeljka koriste se za inicijalizaciju novog klastera. Ako Patroni “prikačimo” na postojeću instancu PostgreSQL, samo odeljak *dcs* ostaje relevantan i čuva se u etcd skladištu za dalju upotrebu. *Dcs* deo odnosi se na definiciju klastera. Počinjemo postavljanjem *ttl* na 30, što znači da primarni čvor mora da povrati svoj status svakih 30 sekundi ili da potencijalno pokrene preusmeravanje na drugi čvor. Postavljanjem *loop\_wait* na 10, replika bi trebalo da primeti master koji nije dostupan za 10 sekundi ili manje. Postavljanje parametra *retry\_timeout* na 10 u suštini sprečava zaustavljene veze tokom operacija, odnosno ako serveri nestanu zbog mrežnih problema. I na kraju, postavili smo *maximum\_lag\_on\_failover* na

bajtni ekvivalent od 1 MB, što je minimalni prag koji replike moraju da zadovolje pre nego što se smatraju kandidatima za čvorove koji nisu dostupni.

Nakon osnovnih dcs elemenata, definišemo kako Patroni obrađuje PostgreSQL. Ako koristimo PostgreSQL 9.5 ili noviju, možemo da postavimo *use\_pg\_rewind* na true kao brži metod za transformisanje nekadašnjeg primarnog u novu repliku bez potrebe za ponovnom sinhronizacijom podataka. Takođe preporučujemo upotrebu slotova za replikaciju u PostgreSQL 9.4 i novijim verzijama kada je moguće sprečiti nenadoknadivo zaostajanje replike i postavljanje use\_slots na true čini to eksplicitnim za Patroni.

Pododjeljak *parameters* je samo niz vrednosti koje se obično nalaze u postgresql.conf. Verzije PostgreSQL starije od 12 mogu takođe da odrede odeljak *recovery\_conf*, koji odgovara datoteci *recovery.conf* koja se koristi za definisanje operacija oporavka replike. To su posebno dopunske vrednosti, što znači da ih dodajemo samo ako želimo da zamenimo podrazumevane vrednosti Patroni ili da definišemo bilo koje konfiguracione elemente za koje smatramo da su kritični za operaciju klastera.

Nakon odeljka *dcs* dolazi odeljak **initdb**, koji se u osnovi koristi za rukovanje parametrima programa *initdb* PostgreSQL-a. U ovom slučaju, odlučili smo da omogućimo kontrolne sume podataka i osiguramo da novoinicijalizovana baza podataka koristi kodiranje znakova UTF8.

Navođenje ove poslednje vrednosti možda se ne čini potrebnim, ali postoje na baze podataka kodirane ASCII-om i vrlo je teško popraviti ih kada se uspostave. To se ponekad može dogoditi ako lokalizacija mašine nije pravilno pokrenuta, kao što je to ponekad slučaj u Docker ili LXC kontejnerima.

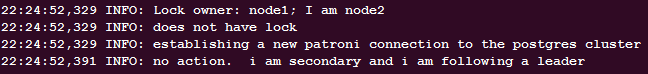
Zatim postoji odeljak *pg\_hba* za dodatne unose u nove datoteke pg\_hba.conf. U ovom slučaju, odlučili smo da dozvolimo da *rep\_user* nalog kontroliše replikaciju pseudo-baze podataka, a svi ostali nalozi mogu se povezati unutar naše ograničene podmreže. Tu biste stavili sve potrebne unose pg\_hba.conf za osnovnu operaciju klastera u stek aplikacija. Dalje, imamo odeljak za korisnike, gde možemo stvoriti onoliko korisničkih naloga za novopokrenuti klaster koliko nam treba. U našem slučaju odlučili smo se za jedan administratorski nalog sa mogućnošću generisanja daljih uloga i baza podataka. Ovaj deo je i razlog zbog kojeg želimo da datoteka bude u vlasništvu korisnika sistema postgres. Želimo da lozinku izložimo što je to manje moguće.

Poslednji odeljak u konfiguracionoj datoteci je **postgresql**, koji određuje stanje svake lokalne PostgreSQL instance. Kao i odeljak *restapi*, i ovde postoje *listen* i *connect* ulazi za definisanje ciljeva mrežnih povezivanja. Ovde takođe definišemo *data\_dir* jer se podaci PostgreSQL mogu nalaziti na različitim lokacijama na svakom serveru. Razlog zbog kojeg smo *bin\_dir* eksplicitno postavili na potpunu putanju binarnih datoteka PostgreSQL-a je zbog mogućnosti servera koji hostuju više verzija PostgreSQL-a ili zbog korišćenja nestandardnih direktorija za instalaciju. Ovde koristimo vrednost koju smo ranije dobili sa *pg\_config*.

Pasus *pgpass* i *authentication* u suštini idu zajedno. Prvi definiše lokaciju za privremenu datoteku lozinke, a drugi deklariše i replikaciju i superuser-a. Budući da je neophodna odgovarajuća potvrda identiteta da bi se nedavno pripremljene replike same pokrenule i započele replikaciju, ovde osiguravamo da postupak uvek uspe.I na kraju, možemo pružiti onoliko proizvoljnih vrednosti postgresql.conf u sekciji parametara koliko želimo. Za razliku od onih u *bootstrap*, oni se primenjuju samo na instancu kojom upravlja trenutni Patroni čvor.

Kada konfiguracionu datoteku učinimo čitljivom za korisnika postgres-a, možemo pokrenuti Patroni na svakom čvoru prosleđujući punu putanju  konfiguracione datoteke naredbi patroni.

Nakon što startujemo servis trebalo bi da dobijemo izlaz nalik ovom:



Slika 8 - Pokretanje Patroni servisa

Kao što smo to uradili sa *etcd* servisom, i Patroni možemo kontrolisati pomoću *systemd*. Za to moramo da kreiramo fajl *patroni.service* na putanji */etc/systemd/system, n*akon čega servisom možemo upravljati pomoću komandi:

*sudo systemctl enable patroni*

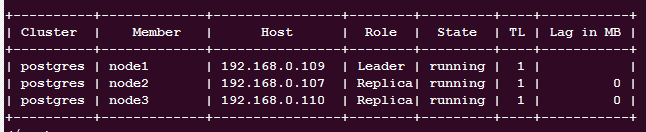
*sudo systemctl start patroni*

*sudo systemctl stop patroni*

Status klastera možemo proveriti komandom:

patronictl -c /etc/patroni.yaml list

i dobićemo sledeći izlaz:

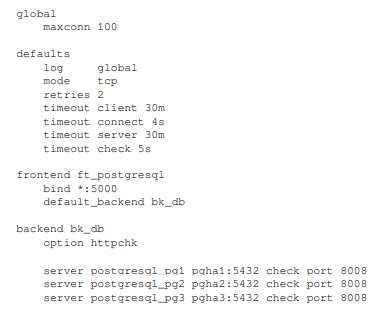


Slika 9 - Status Patroni klastera

## Konfigurisanje HAProxy komponente

Poslednji element steka je HAProxy. Patroni ga koristi za preusmeravanje saobraćaja na primarni čvor u našem PostgreSQL klasteru. Tehnički, ova komponenta nam nije neophodna, jer će Patroni raditi i bez nje. Međutim, ako želimo da sposobnost uvek dođemo do primarnog čvora bez obzira na njegovu lokaciju, onda je od suštinskog značaja.

Za početak, potrebno je kreirati fajl *haproxy.cfg* na putanji /*etc/haproxy,* sa sadržajem kao na slici:



Slika 10 - Konfiguracioni fajl HAProxy instance

HAProxy ima vrlo moćnu konfiguracionu sintaksu potpomognutu stotinama parametara. Zbog toga pisanje konfiguracione datoteke od nule može biti izuzetno težak posao. U našem slučaju, broj parametara koje treba da postavimo je zapravo prilično minimalan.

Počinjemo postavljanjem ***maxconn*** parametra na 100 veza. Ovo je broj veza kojima će HAProxy upravljati pre nego što im jednostavno dozvoli da stanu u red čekanja kernel baferu. Dalje, postavili smo parametar ***log*** na *global*, tako da sve instance HAProxy pišu logove na isti izlaz. HAProxy je u osnovi HTTP proxy sistem, pa moramo osigurati da je parametar ***mode*** podešen na *tcp*.

Nakon što su postavljene ove osnovne stvari, definišemo broj pokušaja, kao i vrednosti raznih vrsta timeout-a. Svi oni podležu obrascima korišćenja, pa ih slobodno treba modifikovati kako bi bolje odgovarali potrebama klastera. Ako klijent ili server ne rade više od 30 minuta, veza će biti prekinuta, što pokazuju client i server timeout parametri.  Takođe smo postavili vremensko ograničenje veze na 4 sekunde, tako da HAProxy ne čeka zauvek da uspostavi vezu. I na kraju, postavili smo vremensko ograničenje za proveru na 5 sekundi, tako da jednom uspostavljena veza ne živi mnogo duže nego što je potrebno pre nego što se prekine.

Naredno je definisanje ***frontend*** i ***backend*** akcija. HAProxy će na frontend strani upravljati dolazne veze, pa kreiramo novi interfejs pod nazivom ft\_postgresqll. U sklopu ove definicije postavili smo *bind* na \*: 5000 da slušamo sve dostupne interfejse na portu 5000. Zatim, povezujemo frontend sa backendom koju ćemo nazvati *bk\_db*.

Na bakcendu, HAProxy će prosleđivati veze na naš primarni PostgreSQL server. Da bismo to rešili, kreiramo novi backend pod nazivom *bk\_db*, koji smo se već povezali na frontend. Jedina opcija koju smo postavili ovde *httpchk*, metoda koju bi HAProxy trebalo da koristi za potvrdu ispravnosti servera. Svi ostali redovi u delu za backend odnose se na jedan od naših PostgreSQL servera. Svaka linija servera podeljena je u tri različita dela. Prvo dolazi ime servera, zatim host i port servisa i na kraju dalje opcije za definiciju. Pored kombinacije host: port, za svaki server definisali smo i port za proveru 8008. Ova opcija govori HAProxy-u da se poveže na port 8008 radi određivanja ispravnosti servera. Kada se HAProxy poveže sa serverom na port 8008, zapravo se povezuje sa Patroni.

Budući da svaki lokalni Patroni čvor zna da li je primarni sistem ili ne, HAProxy zapravo pita svaki čvor da li je primarni. To je dobar način da se iskoristi proxy provera ispravnost. Sa ovim konfiguracionim vrednostima, svaka dolazna veza na port 5000 na bilo kom HAProxy hostu biće prosleđena onom serveru koji je prošao sigurnosnu proveru backenda. Zbog ovoga nema potrebe za virtuelnim IP adresama ili CNAME definicijama - uvek ćemo biti upućeni na ispravan sistem.

# Testiranje HA rešenja – primeri

## Promena master čvora(Performing a managed switchover)

Upravljanje Patroni klasterom je relativno lako ukoliko sve funkcioniše normalno. Primarni razlog za to je obezbeđeni *patronictl* alat za komandnu liniju. Pored jednostavnog prikazivanja statusa klastera, on nudi i nekoliko drugih korisnih operacija. Konkretno, možemo ga koristiti za primoravanje primarnog čvora da se “povuče” i dozvoli da jedna od replika zauzme njegovo mesto. U HA kontekstu, ovo je odličan način za update sistema. Tokom ovog procesa baza podataka nikada nije nedostupna. Zbog toga se koristi i za redovno održavanje sistema. Obzirom da su ovo neki od osnovnih slučajeva na koje treba da se obrati pažnja kada se govori o HA, i da se dešavaju jako često, pa su samim tim i jako važni, demonstriraćemo na primeru kako to izgleda.

Na čvoru koji je trenutni master izvršićemo sledeću komandu kao *postgres* korisnik:

***patronictl -d node1:2379 switchover stamped***

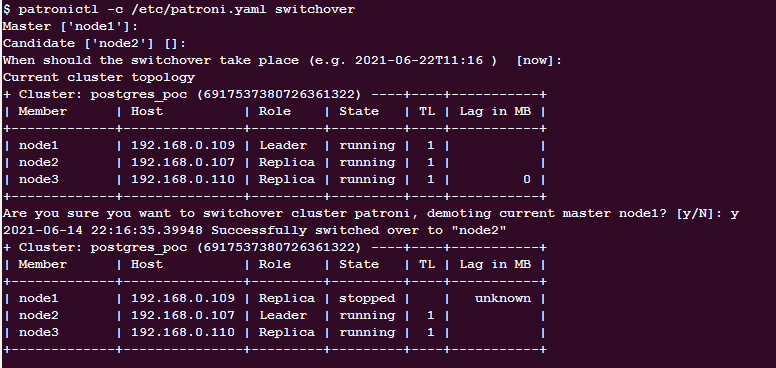
Nakon toga, sačekaćemo nekoliko sekundi i, ponovo kao postgres korisnik, izvršiti:

***patronictl -d node1:2379 list patroni***

Pozivanjem *patronictl* sa parametrom *switchover*, govorimo mu da definitivno želimo da neki drugi čvor “promoviševimo” u master. Flag *-d* omogućava nam da postavimo *etcd* kao izvor konfiguracije i obično je sigurnija opcija, jer uvek treba da odražava trenutno stanje klastera. Naš primer je ciljao server na node1 čvoru, na portu etcd od 2379, ali mogli smo da koristimo bilo koji od servera iz klastera. Budući da distribuirani key-value sloj može važiti za bilo koji broj klastera, takođe moramo odrediti *patroni* kao ime klastera kojim želimo da upravljamo.

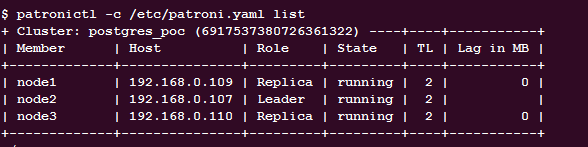
Jednom kada pozovemo naredbu *switchover*, Patroni postavlja više pitanja da verifikuje proces kako bi bio apsolutno siguran pre nego što promeni stanje klastera. Mogli bismo odabrati bilo koju od replika za promociju, ali ako to ne učinimo, Patroni će izabrati jednu u naše ime.

Nakon što potvrdimo poslednje pitanje, Patroni će ovako prikazati izlaz statusa:



Slika 11 - Promena mastera primer

Node1 je sada označen kao zaustavljen - *stopped*, a node2 je novi master u klasteru. Ovo stanje je zapravo samo privremeno. Nikada nismo uklonili node1 iz klastera, pa će ga Patroni modifikovati tako da deluje kao replika. Ako sačekamo nekoliko sekundi i proverimo status klastera parametrom *list* videćemo:



Slika 12 - Status nakon switchover-a

Postupak za povratak prethodnog master-a i njegovo pretvaranje u repliku obično zahteva nekoliko naredbi. Morali bismo ručno da pozovemo *pg\_rewind* ili *rsync*, da pronađemo lokaciju novog master-a, zatim da modifikujemo recovery.conf i ponovo pokrenemo instancu pomoću *pg\_ctl*. Patroni automatski izvodi sve ove korake. Na taj način pomoću Patroni-a dobijamo samoodrživ sistem koji je otporan na greške i ima svojstvo “samoizlečenja”. Takav sistem je jako teško “pokvariti”, čak i namerno, a to je upravo ono što očekujemo kada kažemo high-availability rešenje.

## Prekid usluga servera (outage)

Svaki HA klaster mora imati sposobnost otkrivanja kvara servera i prekida usmeravanja zahteva ka njemu, odnosno sposnobost “zaobilaženja” te instance (route around server failure). Kvarovi, bilo da je njihov uzrok greška na hardveru, greške u radu virtuelnih instanci, pogrešno napisane naredbe, ili nešto četvrto, dešavaju se vrlo često. Najbolji način da utvrdimo pravu otpornost našeg rešenja je da ga testiramo tako što ćemo nešto “pokvariti”.

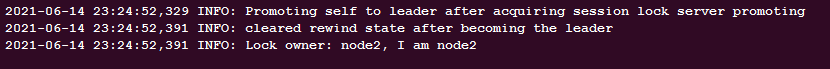
Pod pretpostavkom da su sve tri dosadašnje instance Patroni servera i dalje aktivne, na instanci *node2,* kao korisnik sa *root* privilegijama, uradićemo sledeće:

*sudo systemctl stop patroni*

Logove na instancama *node1* i *node3* pratimo sa: *sudo journalctl -u patroni.*

Ovde se oslanjamo na trik kako bi se izbegao dug i iritantan proces ponovnog pokretanja servera. Patroni daemon sebe postavlja za rukovaoca usluga PostgreSQL servisa. Baš kao što će ponovo pokrenuti PostgreSQL servis koji zaustavimo bez njegove dozvole, isto tako će i zaustaviti PostgreSQL servis ako zaustavimo sam Patroni servis.

Pošto smo Patroni postavili kao sistemski servis, možemo da koristimo naredbu *sistemctl* da zaustavimo demon patroni. Moramo zaustaviti Patroni na serveru koji trenutno deluje kao primarni čvor. Ako se Patroni ne izvršava na instanci koja je master, onda će i *lock* na master čvor u key-value sloju (etcd) isteći. Pri sledećoj internoj proveri statusa, i *node1* i *node3* primetiće da nema registrovanog lidera i pokušaće da zatraže tu poziciju. Samo jedan čvor može pobediti u ovoj trci. Ako posmatramo log sa obe preostale instance, možemo i videti kako izgleda “preuzimanje”:



Slika 13 - Preuzimanje uloge mastera

## Vraćanje instance u klaster

Oporavak sistema nakon većeg pada nije ugodno iskustvo. Moramo da ponovo pokrenemo ili vratimo jedan ili više servera, izvršimo analizu da bismo utvrdili osnovni uzrok kvara i pokušamo da popravimo ili zamenimo neispravne podatake. Ovo generalno važi i za sisteme koji se oslanjaju na Patroni kao svoje rešenje visoke dostupnosti. Međutim, Patroni automatizuje dosadnije delove oporavka oštećene baze podataka PostgreSQL. Na primeru ćemo pokazati kako ponovo pokrenuti prethodno oštećeni ili oflajn PostgreSQL čvor u okviru Patroni klastera.

Pre svega, potreban nam je da dođe do prestanka rada servera u potpunosti. Najlakši način je da to sami “veštački” izazovemo tako što ćemo zadati komande koje će simulirati pad servera od kog se on sam ne može oporaviti. Na jednoj od 3 instance potrebno ja uraditi sledeće:

*pkill -9 patroni*

*pkill -9 postgres*

*find /db/pgdata -name '\*r\*' -o -name '\*0\*' -delete*

To će prouzrokovati “nasilni” prekid rada Patroni i PostgreSQL servisa i ukloniti nekoliko fajlova potrebnih za fukncionisanje PostgreSQL-a. Sada ćemo na istoj instanci uraditi:

1. Uklonimo sadržaj oštećenog klastera pokretanjem sledeće naredbe kao postgres korisnik:

*rm -Rf / db / pgdata*

2. Pokrenemo novi daemon patroni sa ovom komandom kao korisnik sa root privilegijama:

*sudo sistemctl start patroni*

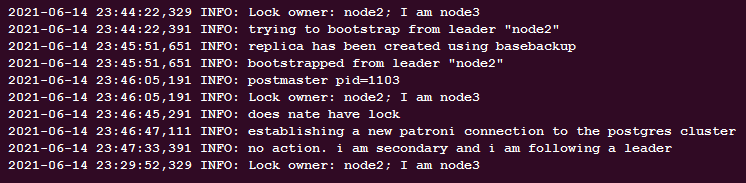
3. Pratimo Patroni log:

*sudo journalctl -u patroni -f*

Ako je pad sistema dovoljno ozbiljan, ne znamo tačno kolika je šteta napravljena na sistemskim fajlovima. Ako naša baza podataka nije inicijalizovana pomoću kontrolnih suma datoteka, mogle bi proći nedelje pre nego otkrijemo neku grešku koja je nastala ranije. Ako server koji je pao preuzeo ulogu primarnog, pre nego što se to dogodilo, ta oštećenja bi se eventualno mogla kopirati na druge instance.

Sigurnije je jednostavno krenuti od nule. Stoga je naš prvi korak brisanje samog direktorijuma **/db/pgdata**. Bez starih datoteka koje bi mogle da utiču na grešku u radu Patroni servisa, Patroni će obnoviti podatke pozivanjem *pg\_basebackup* i konfigurisanjem instance koju smo definisali u */etc/patroni/patroni.yml*. To možemo čak i da vidimo prateći logove.

Ovo  je ono što bi trebalo da  vidimo prilikom ponovnog kreiranja čvora sa praznim **data** direktorijumom:



Slika 14 - Vraćanje instance u klaster

Ovo možda deluje previše lako, ali to je zaista ono što bi trebalo da uradimo - brisanje *data* direktorijuma i pokretanje Patroni servisa. Što je lakši proces, to je teže napraviti grešku, zbog čega je ovaj scenario spada u one koje ističu prednosti korišćenja Patroni rešenja.

## Dodavanje instanci u klaster

Na kraju, možda ćemo odlučiti da proširimo naš klaster PostgreSQL servera kako bismo omogućili više prometa, dodatno povećali dostupnost ili isključili neki od starih sistem. Jednom kada smo uspostavili etcd + HAProxy + Patroni stek, koliko je težak proces dodavanja novih čvorova?

Za demonstraciju dodaćemo *node4* instancu u klaster.Da bismo integrisali novi čvor, treba da dodamo svaku potrebnu komponentu steka. U ovom slučaju, etcd je prvi i najkomplikovaniji deo. Generalno, možemo pratiti postupak instalacije kako je navedeno u delu za Instaliranje i konfigurisanje etcd, ali još uvek ne smemo pokrenuti servis etcd.Prvo šta treba uraditi je sledeće:

*etcdctl member add node4*[***http://node4\_ip:2380***](http://node4_ip:2380)

Posle toga modifikujemo *etcd.conf* na instanci node4 tako da sadrži ove linije:

*initial-cluster-state: existing*

*Initial-cluster: "pgha4=http://pgha4:2380,pgha1=http://pgha1:2380,pgha2=http:// pgha2:2380,pgha3=*[*http://pgha3:2380*](http://pgha3:2380)*"*

Kada smo prvi put pokrenuli etcd, naveli smo *initial-cluster* parametar u konfiguracionojm fajlu. Ovaj parametar nije obuhvatio node4 kada je klaster uspostavljen, tako da etcd neće prepoznati njegove pokušaje da se pridruži. Možemo da izmenimo definiciju klastera pozivanjem naredbe *etcdctl* sa parametrom ***member add***. Treba da dostavimo samo ime i informacije o njegovoj  vezi kako bi etcd znao kako da se poveže sa njim.

Zatim treba da dodamo njode4 u listu servera u parametru *initial-cluster* u sopstvenoj konfiguracionoj datoteci. Ovo omogućava da se on pridruži klasteru na sličan način kao i originalni članovi. Jedina razlika je u tome što takođe moramo da postavimo parametar  *initial-cluster-state* na ***existing*** tako da se etcd na node4 pridruži trenutnom klasteru umesto da kreira novi.

Jednom kada na pravi način dodamo node4 u klaster etcd, sigurno je pokrenuti servis etcd, što možemo učiniti sa *sudo systemctl start etcd* . Posle ovoga, preostali koraci za integraciju su jednostavniji.

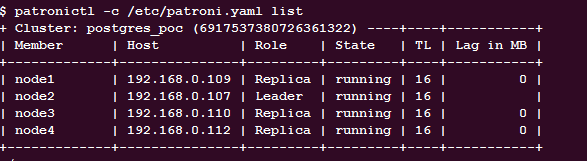
Pošto smo uspešno startovali etcd, na redu je HAProxy. Njega možemo instalirati i konfigurisati baš kao i na ostalim čvorovima. Čak možemo pokrenuti servis bez brige jer se on jednostavno neće povezati ni sa jednom uslugom na novom čvoru dok ne dodamo potrebnu liniju za konfiguraciju servera. U tu svrhu modifikujemo *haproxy.conf* konfiguracioni fajl tako da u *backend bk\_db* sadrži liniju:

*server postgresql\_pg4 pgha4:5432 maxconn 100 check port 8008*

, nakon čega pokrećemo i sam daemon sa *sudo systemctl reload haproxy .*  Tako dolazmo do dela za konfigurisanje Patroni-a na način koji smo već opisali ranije u radu. Patroni će se povezati sa trenutnim masterom, kreiraće novi direktorijum podataka kloniranjem sadržaja sa primarnog čvora i automatski dodati instancu u Patroni sloj.

I za kraj, na bilo kojoj od instanci možemo da pokrenemo komandu za dobijanje statusa klastera

*patronictl -d node:2379 list patroni*



Slika 15 - Status klastera

# Zaključak

High Availability je jedan on najvažnijih koncepta u realnim sistema, a u ovom radu videli smo i zašto. Objasnili smo šta podrazumeva sistem za koji se može reći da je HA. Takođe, videli smo neke od osnovnih pojmova pri radu sa HA sistemima i kako se dostupnost meri. Kao primer jednog takvog sistema videli smo rešenje zasnovano na Patroni-u. Do detalja je objašnjen postupak izgradnje HA rešenja, konfigurisanje svake od komponenti, njihova organozacija u slojevima, koje se njihove uloge i kako se povezuju.

Kao test našeg rešenja, obrađeno je nekoliko najčešćih slučajeva pri radu sa HA sistemima, kako bismo videli da li naše rešenja zadovoljava standarde nekog od pravila broja devetki. Za svaki od testova praćen je status svih instanci u klasteru kako bismo dokazali da nema prekida u radu sistema.

# Literatura

1. The Raft Consensus Algorithm: https://raft.github.io/  [poslednji pristup 22.06.2021.]
2. HAProxy: http://www.haproxy.org/ [poslednji pristup 22.06.2021.]
3. etcd: https://github.com/coreos/etcd [poslednji pristup 28.06.2021.]
4. Patroni: <https://github.com/zalando/patroni> [poslednji pristup 29.06.2021.]
5. psycopg2: http://initd.org/psycopg/ [poslednji pristup 24.06.2021.]
6. PyPI: https://pypi.org/ [poslednji pristup 27.06.2021.]
7. PyYAML: <https://pyyaml.org/> [poslednji pristup 22.06.2021.]
8. Hans-Jurgen Schonig, “Mastering PostgreSQL 11 Expert techniques to build scalable, reliable, and fault-tolerant database applications (Secon edition) “, Packt publishing, October 2018
9. Korry Douglas, Susan Douglas, “PostgreSQL: A Comprehensive Guide to Building, Programming, and administering PostgreSQL databases”, Sams Publishing, February 2003
10. Salahaldin Juba, Andrey Volkov, “Learning PostgreSQL 11 ( Third edition) “, Packt publishing, January 2019