Mehanizmi CC

Veljko Petrović
Novembar, 2022

Mehanizmi Cloud Computing

Upotrebe tehnologije za ostvarnje ciljeva

Logički obod mreže

Virtuelne privatne mreže za upotrebu u oblaku

Šta je logički obod mreže

- Mehanizam logičkog oboda mreže služi da bi računare koji su povezani heterogenim mrežama, neke od kojih su javne, u jednu celinu.
- Od tog povezivanja želimo da dobijemo određene garancije i postignemo neke posebne ciljeve
- Ti ciljevi kao centralnu ideju imaju uspostavljanje *oboda* mreže koji ne postoji u stvarnosti (nisu svi računari u nekakvoj lokalnoj mreži u nekoj sobi negde), no je rezultat korektnog ponašanja mrežnih mehanizama.

Ciljevi logičkog oboda mreže

- Izolacija resursa u oblaku od neovlašćenih korisnika
- Izolacija resursa u oblaku od osoba koje uopšte nisu korisnici
- Izolacija resursa u oblaku od konzumenata usloga oblaka
- Kontrola protočnog opsega dostupnog tim resursima

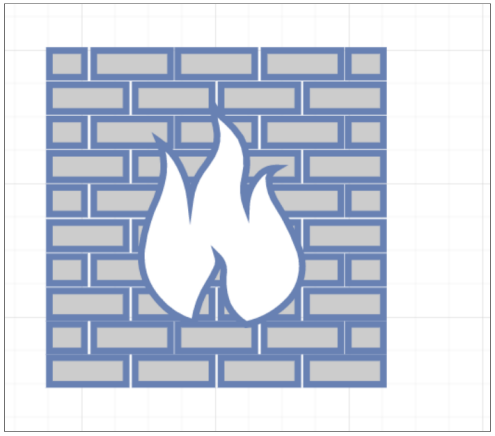
Metode postizanja logičkih oboda mreže

- Virtualizovani firewall
- Virtualizovana mreža

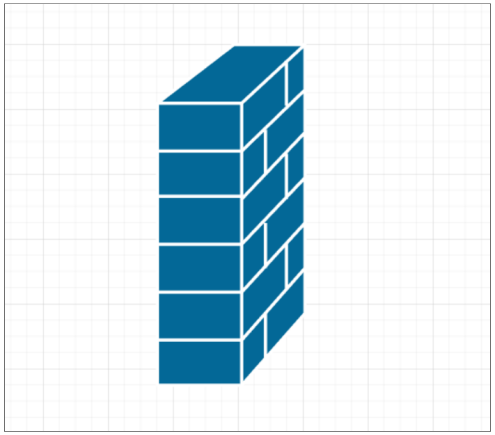
Šta tačno radi ‘vatreni zid’

- Firewall koji se često (i pogrešno) prevodi kao ‘vatreni zid’ je osnovni mehanizam mrežne kontrole
- To je mehanizam koji se postavi na liniju kroz koju saobraćaj mora proći da bi stigao iz jedne mreže u drugu
- Zatim on određenu mrežnu komunikaciju dozvoli, a određenu odbije
- Šta se dozvoli i šta se odbije zavisi od *pravila* firewall mehanizma

Firewall simbol - primer



Firewall simbol - primer



Pravila

- Pravila firewall mehanizma se sastoje, tipično, od karakteristike toka saobraćaja i od akcije
- Karakteristika toka saobraćaja opisuje na koji saobraćaj želimo da se neko pravilo odnosi
- Akcija je ono što želimo da bude odrađeno sa nekim saobraćajem

Tokovi saobraćaja

- Fundamentalno, svi podaci koji dolaze preko mrežnog interfejsa vašeg računara su jedan beskonačan niz jedinica i nula u kome je sve pomešano.
- Kao što možete predpostaviti, ovo nije osobito koristan način da se posmatra saobraćaj, stoga se taj niz jedinica i nula u skladu sa raznim mrežnim protokolima particioniše po različitim osobinama: usmerenju, sadržaju, metapodacima...
- Kroz specificiranje određenih opsega mogućih vrednosti, možemo da definišemo jedan specifičan ili porodicu povezanih tokova saobraćaja

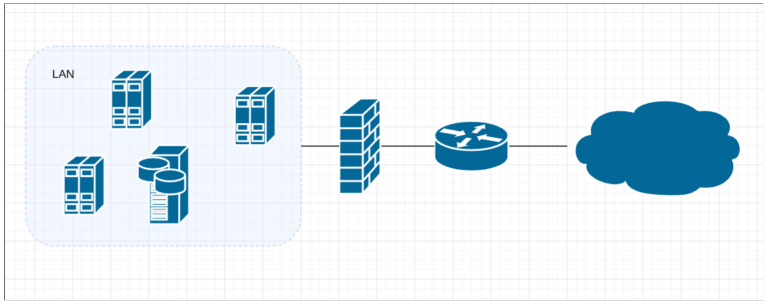
Kako karakterišemo tokove saobraćaja

- Interfejs
- Odredište
- Izvorište
- Port
- Prethodni kontekst (povezani i nepovezani saobraćaj)
- QoS
- Sadržaj (“dubinska inspekcija”)

Uloga firewall sistema u mrežnim obodima

- Ovaj mehanizam možemo da koristimo da, kada se postaramo da se nekakvi računarski resursi povezuju sa širom mrežom preko jedne tačke da na tu tačku postavimo filter koji određene vrste saobraćaja blokira ili tretira na poseban način.
- Kada bi imali sve računara u nekakvom centru onda je to lako: učinimo da odmah iza rutera stoji firewall koji filtrira saobraćaj
- Takav sistem bi, recimo, blokirao bilo koji pokušaj da se pristupi internim serverima.

Slučaj lokalne mreže



Šta bez luksuza LAN

- Šta da radimo kada nemamo našu lokalnu mrežu, šta kada nam je struktura sastavljena od mnogo lokalnih mreža povezanih sa internetom, a mi treba da nekako filtriramo “spoljašnji” saobraćaj i dozvolimo “unutarnji”
- Šta je ovde “unutra” i “spolja”
- Ranije, fizički i logički koncept “spolja” i “unutra” je identičan
- Sada?

Virtuelne privatne mreže

- Ovo je nekada bio visoko specijalizovan alat koji je danas doživeo slavu kao alat opšte bezbednosti
- Koliko dobro radi kao takav alat je otvoreno pitanje
- Ideja virtuelne privatne mreže je bazirana na konceptu *tuneliranja protokola*

Tuneliranje

- Ideja je sledeća: svaki od nivoa OSI modela je, fundamentalno, sačinjen od nekakvih bitova i prenosi opet nekakve bitove
- Izuimajući naravno sloj 1
- Budući da je to tačno, onda protokol bilo kog sloja može u sebi sadržati, kao svoj sadržaj, bilo koji *drugi* sloj.

Primer: Tuneliranje preko HTTPS-a

- Zamislite da vam je, zbog nekog razloga, dozvoljen pristup spoljašnjem svetu isključivo preko HTTPS protokola: imate pristup Web-u, ali ne i bilo čemu drugom, a želite da se povežete na vaš kućni računar koji je vidljiv internetu i ima otvoren port 22 na koji vi hoćete da se zakačite.
- Procedura onda može biti ovakva: Modifikujete vaš operativni sistem (koristite, naravno, Linux) tako da imate instaliran drajver koji predstavlja zamišljenu mrežnu karticu: što se ostatka sistem tiča to *jeste* obična mrežna kartica.
- Taj drajver, umesto da upravlja hardverom, napravi umesto toga HTTPS konekciju na server koji vi specificirate.

Primer: Tuneliranje preko HTTPS-a

- Ta HTTPS konekcija je inače normalna, ali umesto da šaljete obične zahteve, vi umesto toga uvek šaljete zahteve tipa POST čiji je sadržaj podataka bit po bit kopija svih paketa koja vaša virtuelna mrežna kartica dobija.
- Pravi drajver mrežne kartice bi uzima iste te podatke i pakovao ih u Ethernet frejmove i onda slao: vi ga pakujete u POST ‘frejmove’ i šaljete dalje.
- Odgovore koje dobijate vi dekodirate na isti način: predpostavljate da su POST zahtevi koje vi dobijate puni bit-po-bit mrežnih podataka koje dekodirate i šaljte operativnom sistemu kao da su upravo stigle preko mrežnog interfejsa.

Primer: Tuneliranje preko HTTPS-a

- Vodite računa da ovo ima nekakva pojednostavljenja
- Kako operativni sistem razume drajvere je nešto kompleksnije
- Takođe, u praksi bi verovatno želeli *dve* HTTPS konekcije u jednoj gde vi šaljete, a druga gde ste server da bi imali potpun dupleks

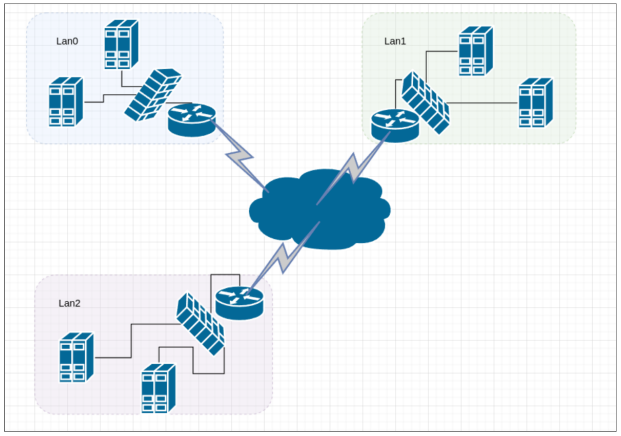
Primer: Tuneliranje preko HTTPS-a

- Na vašem kućnom računaru uradite sličnu stvar: promenite drajver, napravite virtuelnu mrežnu karticu, i dekodirate podatke kao da su običan mrežni saobraćaj.
- Nad ovim parom virtuelnih mrežnih kartica vi možete da uspostavite klasičnu IP vezu sa klasičnim adresama i da konfigurišete rutiranje tako da paketi koji dolaze preko te virtuelne mrežne kartice budu usmereni ka lokalu ili čak rutirani dalje prema internetu ili vašoj kućnoj mreži.
- Ako uradite sve to, onda imate virtuelnu privatnu mrežu.

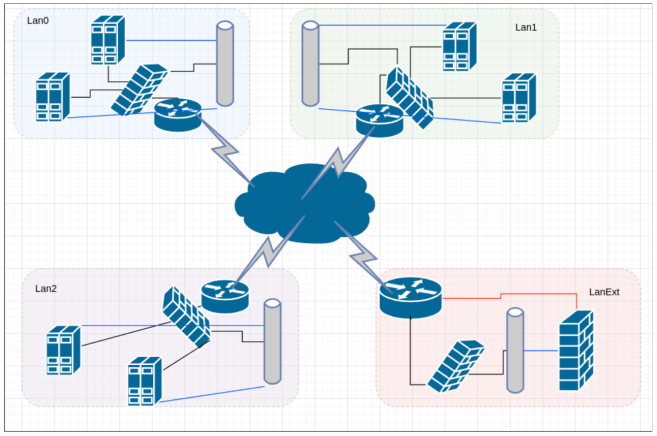
Virtuelne privatne mreže u CC

- Zamislite situaciju koju smo već opisali: imate razne privatne mreže po raznim mestima, sve povezane na internet i vi hoćete ovome da nametnete ‘unutra’ i ‘spolja’
- Kako?

VPM i virtuelni obod - stanje pre



VPM i virtuelni obod - rešenje



VPM i virtuelni obod

- Šta ovaj dijagram u stvari predstavlja?
- Imamo tri mreže koje su povezane lokalno i sa ruterom kroz firewall
- Taj firewall je namešten da blokira sav saobraćaj (čineći mrežu bezbednom) osim saobraćaja koji ga povezuje sa VPM simbolom (cev)
- Taj saobraćaj i samo taj je dozvoljen

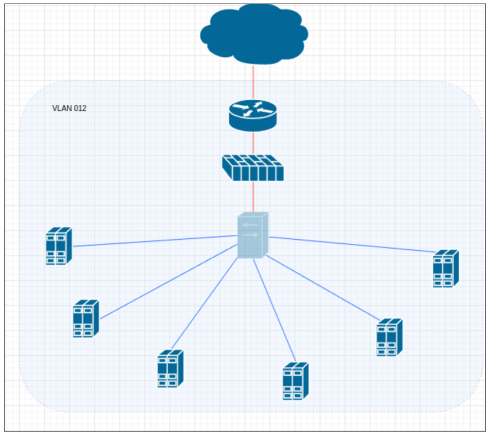
VPM i virtuelni obod

- VPM za računare u LAN0, LAN1, i LAN3 je kao lokalna mreža, ali zahvaljujući tuneliranju, tu mogu da vide i druge računare, kao da su prisutni lokalno.
- Saobraćaj ide kroz otvoreni internet ali enkripcija tačka-do-tačka saobraćaja ga štiti: tj. ceo paket koji putuje je ceo šifrovan sve sa zaglavljem: kada smo maštali o problemu sa HTTPS-om, tako smo to radili.
- Računari u virtuelnoj mreži vide osim samo sebe i LanExt mrežu u kojoj samo postoji firewall koji oni vide i koji funkcioniše i kao ruter

VPM i virtuelni obod

- Taj firewall je tako namešten da kada dobija saobraćaj preko svog virtuelnog interfejsa da ga rutira prema spoljnjem svetu (crvena linija)
- Time, unutarnji računari mogu dobiti pristup otvorenom internetu ali samo na način koji je grupno filtriran u skladu sa tim kako je već potrebno, dok isti taj internet koriste da komuniciraju jedni sa drugima bezbedno preko interneta.
- Drugi način da se ovo posmatra jeste da je *logička* mapa mreže koja je ovim kreirana je:

VPM logički dijagram



Virtuelni serveri

- Virtuelni serveri su najosnovniji element Cloud Computing infrastrukture
- Nisu ništa komplikovanije od virtuelne mašine ili kontejnera upotrebljenih za pružanje servisa
- Jedina neophodna komplikacija koja je neophodna da bi virtuelna mašina bila i virtuelni server jeste rutiranje svesno porta.
- Ovakvo rutiranje omogućava da zahtevi upućeni *nekom* serveru stignu na neki specifični virtuelni server

Mehanizmi smeštanja podataka

Podaci u oblaku

‘Oblak’ uređaji za smeštanje podataka

- Kao što se može virtuelizovati računar viđen kao mehanizam za obradu podataka tako se može i virtuelizovati mehanizam za *smeštanje* podataka.
- Centalna ideja je ista: amalgamacija i podela
- Drugim rečima, prvo treba unificirati raznovrsne resurse za smeštanje podataka, tako da ih treitramo kao jedan veliki disk sa određenim osobinama koje nama odgovaraju.
- Drugo, treba da taj veliki unificirani disk podelimo na odsečke koje odgovaraju sistemima koje koristimo

Smeštanje podataka

- Kao što virtuelizacija može da se uradi na mnogo načina u zavisnosti od raznih faktora, tako može i da se prostor virtuelizuje na mnoge načine.
- Najprostija forma bi mogla da bude da se, recimo:
  - 6 fizičkih diskova (10TB svaki) povezanih za računarski sistem poveže kroz RAID 6
  - Rezultujući virtuelni disk (40TB) se podeli 1 particiju od 30TB i 20 particija od po 512GB
  - Svaka od tih particija onda bude disk za po virtuelnu mašinu, a sve te virtuelne mašine pristupaju particij od 30TB koja služi za čuvanje podataka.

Kompleksniji mehanizmi

- Kompleksniji mehanizmi podržavaju amalgamiranje i deljenje diska kroz mehanizme koje rade preko mreže, a ne samo povezane za jedan računar.
- Ti sistemi mogu biti posebni i namenjeni paš ovakvim zadacima, ali mogu biti i vrlo jednostavni mehanizmi mrežnog deljenja podataka kao što je recimo NFS za Linux ili SMB za Windows.
- Sve zavisi od potreba, arhitekture, i *nivoa pristupa.*

Nivo pristupa

- Nivo pristupa jeste nivo na kojem se podaci na virtuelzovanom sistemu za deljenje podataka dele/amalgamiraju
- Generalno to znači da se podaci tretiraju lokalno do tog nivoa a virtuelizovano preko tog nivoa
- Možete misliti o tome kako, na primer, možemo imati virtuelni disk ili virtuelnu bazu podataka

Nivoi pristupa

1. **Blokovi.** Najniži mogući sloj deljenja podataka gde se virtualizuje pristup individualnim blokovima. Efekat je virtualizacije *hard diska* kao takvog.
2. **Fajlovi.** Podaci su grupisani u fajlove u direktorijumima: efekat je virtuelizacija sistema fajlova.
3. **Setovi podataka.** Podaci su grupisani u set podataka organizovan u tabele ili kakav drugi sličan sistem. Efekat je virtualizacija podataka kroz bazu podataka.
4. **Objekti.** Podaci su grupisani u objekte slobodnog formata koji su dostupni kroz resursni sistem sličan Web/URI modelu.

Performanse i nivo pristupa

- Generalno govoreći, nivo pristupa baziran na fajlovima se ne preporučuje za performantno usluživanje u oblaku.
- Fajlovi su naravno vrlo generalan način pristupa podacima što može biti zgodno, ali to takođe znači da, efektivno, pristupamo podacima koji su deljeni kao fajlovi, ali su *smešteni* u skladu sa lokalnim pravilima.
- To znači da su osobine fajl sistema prilagođene predpostavkama lokalnog pristupa umesto da, npr. budu optimizovano distribuirane.

Setovi podataka i baze podataka

- Virtualizaciaj kroz baze podataka je veoma pogodna za razvoj aplikacija zato što je ono za šta bi deljeni prostor bio iskorišćen u velikom broju aplikacija je baš nekakva baza podataka
- Dodatna prednost jeste što baze generalno znaju da će biti deljene tako da je optimizacija za to već ugrađena, kao i softver za efektno pretraživanje.
- Najveći izbor ovde je između tradicionalnih relacionih baza i baza koje koriste modele različite od relacionog, poznate žargonski kao NoSQL baze.

Prednosti i mane relacionog modela

- Relacioni model baza podataka je strahovito moćan i performantan naročito ako su podaci rigidno struktuirani, te je moguće napraviti sistem ograničenja koji adekvatno opisuje podatke kroz sistem povezanih tabela (relacija).
- Ovo pruža ovakvim sistemima jako dobre performanse i to je jedan od razloga zašto je ovakav stil baze podataka bio dominantan toliko dugo.
- Problem sa tačke gledišta rada u ‘oblaku’ je teškoća skaliranja ovakve baze.

Horizontalno skaliranje relacionih baza podataka

- Horizontalno skaliranje (za razliku vertikalnog) je bazirano na tome da se posao podeli između više instanci
- Ako je potrebno samo *pristupati* podacima, onda je horizontalno skaliranje trivijalno i sastoji se od potpune replike podataka koja se deli između individualnih servera i gde se zahtevi za podacima distribuiraju korišćenjem klasičnog ‘load balancing’ pristupa kao da je u pitanju npr. web server.
- Kada je pisanje u pitanju nastane ozbiljan problem. Što?

Problem sinhronizacije izmene podacima

- Centralni problem je, prvo, dogovor oko toga čija je ‘starija’ kada je u pitanju promena baze
- Ovo se može rešiti kroz dva procesa: vođenja računa o tome kada se desi koja promena kombinovano sa ‘resorima’ gde je svaki server naročito odgovoran za neki deo neke tabele.
- Budući da su velike tabele u stvari prilično neefikasne, već postoje takvi mehanizmi.
- U PostgreSQL, recimo, postoji mehanizam particionisanja koji omogućava da se jedna tabela razbije na N pod-tabela koje se videkao jedna velika, logička tabela.

Povezanost podataka

- U tom slučaju, moguće je napraviti replikaciju podataka u kome individualni serveri opslužuju specifične particije i onda uraditi replikaciju na takav način.
- Ovo je odista moguće, ali naleće na problem u kome particije nisu nužno *nezavisne.*
- Cela ideja relacionih podataka jesu baš njihove relacije: svaka baza relaciona baza podataka, ako je dobro dizajnirana, se sastoji od relativno prostih tabela povezanih relativno kompleksnim vezama koje omogućavaju da se od njih stvore razni potrebni 'pogledi' na podatke.
- Baš takva dobra baza ima veliku povezanost podataka.

Povezanost podataka

- Što bi povezanost bila problem?
- Centralna teškoća leži u tome što povezanosti idu između particija podataka što znači da se bilo koja promena mora replicirati i kaskadirati između svih čvorova u mreži što, pak, znači da mrežna kašnjenja i zagušenja 'pojedu' sve performanse koje smo dobili replikacijom.
- Kao rezultat, u zavisnosti od baze može biti *nemoguće* da se efektno horizontalno skalira relaciona baza.

'NoSQL' i replikacija

- Kada su upiti u baze u pitanju, SQL je toliko dominantan da je čitav koncept baze koja *nema* relacionu strukturu prozvan 'NoSQL'
- Ideja je da se koristi nekakav drugi model koji je lakši za skaliranje
- Tipičan model je model *dokumenta.*
- Baze podataka dokumenta čuvaju u sebi podatke kao manje ili više celovite *dokumente* koje možete zamisliti kao XML ili JSON zapise (ponekad oni baš to i jesu).

Što je model dokumenta bolji?

- Umesto n-torki povezanih u komplikovanije strukture koje zahtevaju kaskadno pisanje, ovde imamo dokumente koji i dalje mogu biti povezani (MongoDB, recimo, zove metode povezivanja 'view') i kaskadno pisani, ali manje često.
- Naročito ako se koristi mehanizam podele (koji ovakve baze podržavaju) gde je podela između individualnih replika (tehnički termin za takve parcijalne replike je 'shard') inteligentna i bazirana, npr. na tome kakvi podaci idu zajedno.

Mehanizam osmatranja sistema

- Mehanizmi osmatranja sistema su mehanizmi koji se bave očitavanjem tekućeg stanja sistema ne bi li se sistemom bolje upravljalo.
- Naročito ih zanimaju podaci o tome koliko koji sistemi rade, iskorišćenosti resursa, greškama, anomalijama, i bezbednosnim događajima.
- Postoje u dve fudamentarne forme: vođeni događajima i vođeni upitima.

Događaji i upiti

- Mehanizimi vođeni događajima su ne osobito opterećujući sistemi koji ‘osluškuju’ događaje raznih vrsta koji im javljaju individualne mašine i sistemi. Ti događaji se onda agregiraju u čuvaju u bazama podataka logova.
- Mehanizmi vođeni upitom (eng. poll) ne čekaju da se nešto desi, no periodično šalju upite raznim sistemima ne bi li saznali šta se dešava.
- Mehanizmi upita su naročito dobri kada je bitno detektovati kada je neki sistem prestao da funkcioniše ne bi li se pnovo pokrenuo.

Mehanizam replikacije resursa

- Baš kao i virtuelni serveri, ovo je nešto o čemu pričamo od početka i što je osnovna gradivna jedinica sistema u oblaku
- Ideja je veoma jednostavna: da se neka osobina koju želimo može postići tako što imamo veći broj (repliciranih) resursa
- Osobine koje možemo dobiti su, recimo:
  - Performanse (horizontalno skaliranje)
  - Pouzdanost (hot-swap/failover serveri)
  - Bezbednost (redudantna kontrola konsenzusa, npr. konzorcijumski lanci blokova)

Mehanizam već-gotovih okruženja

- Potrebe aplikacija mogu biti predvidive
- Budući da je ovo slučaj, nema potrebe da se mušterije CC rešenja opterećuju konfiguracijom od nule
- Umesto toga, može im se dati i polugotovo okruženje koje samo treba da se podesi
- Ovo se može proširiti u čitav katalog unapred konfigurisanih sistema za razne potrebe

Mehanizam osluškivača telemetrije skaliranja

Upravljanje skaliranjem CC instalacija

Osluškivač telemetrije skaliranja

- Ovo je nezavisni servisni agent koji operiše unutar sistema
- Funkcija je da se meri opseg zahteva ka uslugama i modeluje se količina resursa neophodnih da se ti zahtevi servisiraju
- To onda generiše signal upravljanja koji povećava ili smanjuje količinu resursa alociranih, tj. okida mehanizam horizontalnog skaliranja.

OTS primer: AWS Auto Scaling

- Primer za, recimo, Amazon, je AWS EC2 Auto Scaling kombinovan sa Amazon CloudWatch tehnologijom.
- Ekvivalentna rešenja postoje i za druge platforme, ovo je samo reprezentativan primer veoma velike platforme.
- Ova sekcija je bazirana u nekoj meri na zvaničnoj AWS dokumentaciji dostupnoj na **docs.aws.amazon.com**.

OTS primer: AWS Auto Scaling

- CloudWatch (CW) prati saobraćaj (CW takođe obavlja usluge i drugih mehanizama koje smo pominjali)
- Auto Scaling he mehanizam gde možete skalirati vaše grupe za skaliranje na gore ili na dole rukom ili to učiniti *dinamički* ili *prediktivno.*

OTS dinamičko skaliranje

- Dinamičko skaliranje omogućava da se napravi, efektivno, jednostavna automatika koja povezuje nege ulazne signale sa signalom upravljanja.
- Kada se koristi dinamičko skaliranje na raspolaganju je nekoliko mogućih konfiguracija:
  - Prosto
  - Koračno
  - Vođeno metom

CW Alarmi

- I prosto i koračno skaliranje u AWS okruženju se bazira na alarmima CW mehanizma
- Alarm je jednostavno događaj koji se okine (i koji može da pokrene nekakav menadžment process unutar sistema) kada se neka od metrika koju može da meri CW promeni iznad/ispod nekakve granične vrednosti pod nekakvim okolnostima.
- Alarm, recimo može da bude “Okini ovaj alarm kada je prosečna upotreba procesora preko 80% 4 minuta zaredom.”
- Generalno nam trebaju dva alarma: jedan da aktivira skaliranje na gore i jedan na dole.

Prosto skaliranje

- Prosto skaliranje je bazirano na jednostavnom odgovoru na događaj
- Kada god se desi jedan od dva događaja ili se za neku fiksnu količinu poveća broj instanci ili se smanji
- Padanje u beskonačan jitter oko nulte tačke sprečava upotreba prostog histerezisa kroz tajmer
- Tajmer sprečava da se skaliranje desi dok ne otkuca do kraja

Prosto skaliranje - primer

```
aws autoscaling create-auto-scaling-group --auto-
scaling-group-name moja-grupa \
  --launch-template LaunchTemplateName=moj-
sablon,Version='2' \
  --vpc-zone-identifier "subnet-5ea0c127,subnet-
6194ea3b,subnet-c934b782" \
  --max-size 5 --min-size 1
```

Prosto skaliranje - primer

- Launch Template je mehanizam koji definiše *čime* se proširuje skup računara
- Primetite da ovde definišemo maksimum i minimum: ne bi valjalo da se zbog 5 minuta bez zahteva aplikacija samoisključi niti bi valjalo da vas navala korisnika bankrotira.
- VPC identifikatori nisu bitni za ovu priču i služe da se u bezbednosnom modelu AWS identifikuje on sa čim se radi

Prosto skaliranje - primer

```
aws autoscaling put-scaling-policy --policy-name
prosto-skaliranje-na-gore \
  --auto-scaling-group-name moja-grupa --scaling-
adjustment 30 \
  --adjustment-type PercentChangeInCapacity
```

Prosto skaliranje - primer

- Terminološki komentar: ovde koristimo skaliranje na gore za povećanje kapaciteta i skaliranje na dole za smanjenje kapaciteta. Termin koji se koristi na engleskom često je “scaling out” i “scaling in” za isti koncept, dakle umesto na gore i na dole se korisite ‘upolje’ i ‘unutra.’
- Ovo je jednostavno povećanje kapaciteta za 30%.

Prosto skaliranje - primer

- Ova komanda će vam vratiti JSON odgovor koji u sebi ima ARN za nju
- ARN je standard koji interno koristi AWS da bi imenovao stvari
- Primer ARN-a za ovakvu neku stvar je

Primer ARN-a

```
{
    "PolicyARN":
        "arn:aws:autoscaling:region:123456789012:scaling
        cbeb-4294-891c-a5a941dfa787:autoScalingGroupName
        grupa:policyName/prosto-skaliranje-na-gore"
}
```

Prosto skaliranje - primer

- ARN će biti potreban da bi se pozvali na ovo podešavanje kasnije i uspešno ga povezali sa alarmom
- Generalno, ARN je mehanizam koji omogućava da se jednoznačno identifikuju stavke u okviru sistema
- Vrednost koju vi vidite, ako ovo budete probali, će naravno biti različita

Prosto skaliranje - primer

- Sada je neophodno razraditi i obrnutu politiku koja propisuje da se resursi dealociraju kada je potreba manja
- Ovde imaju dve fundamentalne promene:
  - uvodi se tajmer vrednost koja će sprečiti fluktuaciju alociranih resursa
  - uvodi se drugačija forma skaliranja gde se ukida jedna mašina umesto da se ukupan kapacitet poveća za 30%

Prosto skaliranje - primer

```
aws autoscaling put-scaling-policy --policy-name
prosto-skaliranje-na-dole \
  --auto-scaling-group-name moja-grupa --scaling-
adjustment -1 \
    --adjustment-type ChangeInCapacity --cooldown 180
```

Prosto skaliranje - primer

- Sledeći korak je definisati tačke gde želimo da aktiviramo naše prosto skaliranje
- Ovde je bitno odabrati pravu metriku
- Ako metrika nije dobro odabrana onda ona neće skalirati prema potrebama sistema te skaliranje neće biti obavljano kako treba

Odabir dobre metrike

- Nema standarda za dobru metriku, ona zavisi od vašeg rešenja
- Glavna stvar koja vam je potrebn jeste da metrika meri nešto što je korelat ograničavajućeg faktora u potrošnji resursa vašeg sistema
- Karakterizacija ograničavajućeg faktora je bitan zadatak u disciplini merenja i predviđanja performansi sistema

Kratko o ograničavajućim faktorima

- Vaš sistem može da zakaže i demonstrira smanjene performanse (što pak onda treba da se leči horizontalnim ili vertikalnim skaliranjem) usled velikog broja ishoda
- Generalno govoreći ovde se koristi engleska terminologija gde se specificira šta je tačno faktor limita i onda se dodaje sufiks ‘-bound’
- Recimo proces koji je CPU-bound je limitiran dostupnim procesorskim vremenom/brzinom

Kratko o ograničavajućim faktorima

- Proces može biti ograničen memorijom, dostupnim performansama korišćene baze podataka, dostupnim resursima za ulaz i izlat, dostupnim resursima za mrežni saobraćaj, kao i egzotičnijim faktorima vezanim za to kako operativni sistem apstrahuje pristup hardveru i drugim svojim resursima
- Egzotična forma ograničavajućeg faktora koja može predstavljati bezbednosnu manu jeste ograničenje dostupnom entropijom u generaciji kriptografski-bezbednih slučajnih brojeva.

Interrelacija ograničavajućih faktora

- Kada se radi sa organičavajućim faktorima bitno je uzeti u obzir da je situacija retko takva da je samo jedan faktor pod značajnim naponom
- Iako će jedan faktor biti prvi iscrpljen, te će ograničiti rast performansi, šanse su da će biti opterećeni i drugi faktori
- Kao rezultat, ne mora se meriti baš onaj ograničavajući faktor: samo nešto što se lako i dobro meri i što je dobro korelirano sa onim faktorom koji odista dovodi do zagušenja

Odabir korelata

- Moguće je pogoditi šta će biti ograničavajući faktor u nekom sistemu
- Za neki tipičan back-end to će najverovatnije biti ulaz/izlaz neke prirode, sa kojim će biti razumno dobro korelirana potrošnja CPU-a zato što je akt parsovanja poruka nešto što zahteva anganžman procesora
  - Računanje heš vrednosti ključeva
  - Provera kriptografije na HTTPS zahtevu plus eventualno na potpisima koji su deo samog zahteva ili autorizacionog tokena
  - Obrada podataka koji su stigli
- Ako su performanse bitne ništa ne može da pobedi merenje performansi, potrošnje resursa i njihove korelacije

Podešavanje alarma u prostom slučaju

```
aws cloudwatch put-metric-alarm --alarm-name prosto-
skaliranje-na-gore-alarm \
  --metric-name CPUUtilization --namespace AWS/EC2 --
statistic Average \
  --period 120 --evaluation-periods 2 --threshold 60
\
  --comparison-operator GreaterThanOrEqualToThreshold
\
  --dimensions "Name=AutoScalingGroupName,Value=moja-
grupa" \
  --alarm-actions
arn:aws:autoscaling:region:123456789012:scalingPolicy:a
cbeb-4294-891c-
a5a941dfa787:autoScalingGroupName/moja-
grupa:policyName/prosto-skaliranje-na-gore
```

Podešavanje alarma u prostom slučaju

- Ovo za ranije generisanu grupu napravi alarm koji se okine kada je vrednost izmerena kao prosek u dva perioda merenja od 120 sekundi upotrebe procesora je veća od 60% i pozove politiku za skaliranje na gore tako što se referiše na njen ARN
- U budućim primerima umesto vrednosti ARN-a stavićemo samo `$ARN` da radi efikasnosti notacije

Podešavanja alarma za skaliranje na dole

```
aws cloudwatch put-metric-alarm --alarm-name prosto-
skaliranje-na-dole-alarm \
  --metric-name CPUUtilization --namespace AWS/EC2 --
statistic Average \
  --period 120 --evaluation-periods 2 --threshold 40 \
  --comparison-operator LessThanOrEqualToThreshold \
  --dimensions "Name=AutoScalingGroupName,Value=moja-
grupa" \
  --alarm-actions $ARN
```

Problemi sa prostim sistemom

- Ovo je prilično slaba metoda skaliranja zato što ne ume da reaguje efektno na nagle skokove u zahtevima i oslanja se na tajmere da bi rešila problem fluktuacija.
- Bilo bi bolje imati malo napredniju formu upravljanja
- Jedan metod koji to omogućava donekle jeste da umesto da se reaguje sa samo jednim mogućim odgovorom da se reaguje sa više u zavisnosti od toga koliko je metrika skočila.

Upravljanje bazirano na koracima

- Namešta se vrlo slično kao i jednostavno upravljanje sa jednom glavnom razlikom: politika sada ima različite reakcije na različit porast opterećenja
- Ovo se podešava kada se stvara pravilo skaliranja

Primer upravljanja sa koracima

```
aws autoscaling put-scaling-policy \
  --auto-scaling-group-name moja-grupa  \
  --policy-name korak-skaliranje-na-gore \
  --policy-type StepScaling \
  --adjustment-type PercentChangeInCapacity \
  --metric-aggregation-type Average \
  --step-adjustments
MetricIntervalLowerBound=0.0,MetricIntervalUpperBound=1
\

MetricIntervalLowerBound=15.0,MetricIntervalUpperBound=
\

MetricIntervalLowerBound=25.0,ScalingAdjustment=30 \
  --min-adjustment-magnitude 1
```

Primer upravljanja sa koracima

- Ovde je situacija takva da se u zavisnosti od magnitude prebacivanja preko podešene metrike (iste one kao i ranije) kapacitet povećava za ili 10%, 20%, ili 30% sa minimalnom deltom od jedne instance
- Ovo znači da je odgovor na promenu ponašanja mnogo više adaptabilan

Primer upravljanja sa koracima

```
aws autoscaling put-scaling-policy \
  --auto-scaling-group-name moja-grupa  \
  --policy-name korak-skaliranje-na-dole \
  --policy-type StepScaling \
  --adjustment-type ChangeInCapacity \
  --step-adjustments
MetricIntervalUpperBound=0.0,ScalingAdjustment=-2
```

Primer upravljanja sa koracima

- Ovde se detektuje slučaj da je potrošnja pala ispod nekog praga i odma se smanjuje broj instanci za 2
- Ovo je kombinacija relativno glatkog dodavanja kapaciteta i naglog uklanjanja u slučaju pada broja zahteva

Da li može bolje?

- Ovo je i dalje relativno grubo podešavanja kompleksnog ponašanja
- Ako razmsilite mi na osnovu nekog signala podešavamo upravljački signal gore ili dole
- Ovo je čist problem za automatiku koja zna štogod više od ‘uključi/isključi’ odziva
- Stoga AWS podržava i odziv vođen metama

Upravljanje vođeno metom

- U srži ovo je tehnika gde se specificira željena vrednost neke metrike i onda sistem sam generiše alarme i politike koje skaliraju (u nekim okvirima) sistem tako da se ta željena vrednost održi
- Ovo je klasično upravljanje sa povratnom spregom

Metrike upravljanja

- Metrike koje se mogu ciljati dolaze iz liste unapred definisanih, ali mogu se podešavati i posebno
- Većinu vremena ugrađene metrike su dovoljne
- Ono što karakteriše metrike vezane za upravljanje vođeno metom jeste to što se često odnose na celu grupu u agregatu

Ugrađene metrike

- `ASGAverageCPUUtilization` — Prosečna potrošnja procesora za celu grupu.
- `ASGAverageNetworkIn` — Prosečni broj bajtova koji prima jedna instanca na svim mrežnim interfejsima.
- `ASGAverageNetworkOut` — Prosečni broj bajtova koji šalje jedna instanca na svim mrežnim interfejsima.
- `ALBRequestCountPerTarget` — Prosečan broj balansiranih (više o tome kasnije) zahteva po meti.

Odabir vrednosti metrike za metu

- Postavlja se pitanje kako odabrati željenu vrednost: na kraju krajeva nama neka iskorišćenost nekog resursa nikad nije cilj
- Kriterijum odabira vrednosti se vodi sa dva faktora koji imaju oprečno dejstvo:
  - Ušteda
  - Bafer za nagli porast iskorišćenosti

Ušteda

- Što je veća meta to će da sistem više da proaktivno gasi instance, što pak znači da će ukupna cena održavanja infrastrukture biti znatno manja
- Sa ove tačke gledišta, mi bi voleli da je metrika što veća moguća: recimo 100% za procesor
- To bi značilo da baš ništa nije ‘bačeno’ to jest, da nikada ne plaćamo za neiskorišćene računarske resurse.

Bafer za nagli porast iskorišćenosti

- Bilo koja aplikacija koja se izvršava u CC okruženju može da bude meta naglog porasta iskorišćenosti (eng. *spike*)
- Kada se to desi, politika skaliranja koju smo definisali može da popravi situaciju, ali ovo traje neko vreme: tokom tog vremena sistem će u najboljem slučaju manifestovati lošu latenciju, a u najgorem zahtevi neće biti obrađeni.
- Ovo se može izbeći tako što se alocira više resursa nego što je neophodno: tada kada naglo poraste potreba neophodni resursi su već spremni i mogu da izdrže dok se ne završi horizontalno skaliranje.

Magični brojevi

- Uzimajući u obzir ova dva faktora, dakle, meta treba da je negde između
- Tačno gde zavisi od gomile faktora:
  - Frekvencija naglih porasta iskorišćenosti
  - Intenzitet naglih porasta
  - Važnost garantovanja blagovremenog odziva
  - Vreme skaliranja
  - Ekonomska pitanja

Magični brojevi

- Jedini pouzdan način da se ovi brojevi nabave i podese jeste da se sistem u toku faze testiranja, javne bete, itd. izloži ekstremno detaljnoj analizi svih podataka
- Svi ovi faktori su specifični za pojedinačne aplikacije u specifičnim okolnostima i ne mogu se pouzdano predvideti: moraju se izmeriti.
- Zato je neohodno, kada se upravlja ovakvim sistemima, konzistentno pratiti sve pokazatelje i biti upoznat sa statističkim metodama neophodnim da se iz šume log-ova proizvedu vredni uvidi.
- U nedostatku informacija, neka bezbedna forma margine je otprilike 50-60%

Vreme skaliranja

- Pomenuli smo da je vreme skaliranja bitan faktor u izboru mete: šta je to?
- Vreme skaliranja je vreme neophodno da se novopokrenuta instanca dovede u stanje da je spremna da opslužuje zahteve
- Sistem može da detektuje da se sistem pokrenuo, ali je malo teže znati da li je aplikacija koja se u tom okruženju izvršava uspešno prošla kroz sve stadijume samoosposobljavanja
- Zato postoji vreme skaliranja (eng. warmup time) ne samo kao faktor nego kao i parametar kojim se konfiguriše skaliranje

Vreme skaliranja i automatsko skaliranje

- Upravljanje 'želi' matematički govoreći da bude kontinualno i bavi se infinitesimalnim promenama u infinitesimalnom vremenu
- Ovde radimo sa vrlo diskretnim akcijama koje se tako ne mogu posmatrati
- Tajmer (eng. cooldown) je jedna mera koju koristimo da ispeglamo ovaj problem
- Vreme skaliranja je druga: ona se stara da za neko specificirano vreme novoporeknute instance se računaju kao još isključene za potrebe algoritma za upravljanje.

Podešavanja upravljanja vođenog metom

- Prvi korak je da se generiše JSON fajl koji konfiguriše željenu metu
- Taj fajl mora da opiše koja metrika se posmatra i koje ja željena vrednost
- U našem primeru ciljamo intragrupnu iskorišćenost procesora od 40%

Podešavanja upravljanja vođenog metom

```
{
  "TargetValue": 40.0,
  "PredefinedMetricSpecification":
    {
      "PredefinedMetricType":
        "ASGAverageCPUUtilization"
    }
}
```

Šta da nećemo predefinisanu metriku za metu?

```
{
   "TargetValue":40.0,
   "CustomizedMetricSpecification":{
      "MetricName":"MojaMetrika",
      "Namespace":"ProstorImena1",
      "Dimensions":[
         {

        "Name":"DimenzijaPoKojojSePosmatraRecimoGrupa",
            "Value":"VrednostKojaSePosmatraRecimoCPU"
         }
      ],
      "Statistic":"Average",
      "Unit":"Percent"
   }
```

Kompleksne nepredefinisane metrike

- Ponekad želimo ne samo nešto specifično posmatrano preko neke dimenzije, kao ovde, nego hoćemo i da možemo da računamo naše mere bazirane na posebnim aritmetičkim izrazima.
- I ovo je podržano preko posebne sintakse konfiguracionog JSON fajla

Kompleksne nepredefinisane metirke

```
{
    "CustomizedMetricSpecification": {
        "Metrics": [
            {
                "Label": "Broj poruka koje čekaju na
        obradu",
                "Id": "m1",
                "MetricStat": {
                    "Metric": {
                        "MetricName":
        "ApproximateNumberOfMessagesVisible",
                        "Namespace": "AWS/SQS",
                        "Dimensions": [
                            {
                                "Name": "QueueName",
```

Specifikacija upravljanja sa metom

```
aws autoscaling put-scaling-policy --policy-name
cpu40-upravljanje-metom \
  --auto-scaling-group-name moja-grupa --policy-type
TargetTrackingScaling \
  --target-tracking-configuration file://config.json
```

Specifikacija upravljanja sa metom

- `config.json` je naravno fajl koji smo napravili sa konfiguracijom
- Pozivanje ove komande će da nam vrati ne samo ARN pravila koje smo upravo definisali, nego i svih alarma koje će to automatski da napravi za nas.
- Ove alarme ne diramo mi: oni su atomatski kreirani i biće automatski i obrisani.

Povratna vrednost


```
{
    "PolicyARN": "arn:aws:autoscaling:region:account-
        id:scalingPolicy:228f02c2-c665-4bfd-aaac-
        8b04080bea3c:autoScalingGroupName/moja-
        grupa:policyName/cpu40-upravljanje-metom",
    "Alarms": [
        {
            "AlarmARN":
        "arn:aws:cloudwatch:region:account-
        id:alarm:TargetTracking-moja-grupa-AlarmHigh-
        fc0e4183-23ac-497e-9992-691c9980c38e",
            "AlarmName": "TargetTracking-moja-grupa-
        AlarmHigh-fc0e4183-23ac-497e-9992-
        691c9980c38e"
        },
```


Prediktivni modeli upravljanja

- Šta je bolje od reagovanja na promenu u zahtevima prema resursima?
- Reagovanje *pre* nego što se promena desila.
- Ovo je povremeno trivijalno: ako radimo reindeksiranje svakih šest sati, onda znamo sigurno da će se zahtevi naglo povećati tačno na svakih šest sati.
- Ponekad predviđanje nije trivijalno, no zahteva analizu obrazaca u podacima bilo kroz ljudsku aktivnost bilo kroz automatske sisteme koji mogu biti napajani konvencionalnim algoritmima ili algoritmima mašinskog učenja.

Primer obrasca

Primer obrasca

- Ovo su trenutni konkurentni korisnici na Steam platformi za period od 11.12.2022. do 14.12.2022.
- Šta možemo da naučimo gledajući ovaj grafikon?
- Kako bi to uticalo na naš dizajn infrastrukture?

Detekcija obrazaca

- Ponekad je dovoljno pogledati kao što je slučaj ovde
- Ponekad su obrasci finiji i zahtevaju ili analizu ili nekakav algoritam dobar u detekciji obrazaca
- AWS nudi jedan takav sistem

Prediktivni modeli u slučaju AWS platforme

- Postoje dve forme predikcije koja se može koristiti u autoskaliranju u okviru AWS: predikcija može da bude informativna ili upravljačka.
- Informativna predikcija samo pokušava da ‘pogodi’ kakvo će opterećenje biti unapred i to beleži: može da se koristi da se oceni model kvaliteta.
- Upravljačka radi skaliranje u odnosu na to

Informativna predikcija

```
aws autoscaling put-scaling-policy --policy-name
cpu40-informativna-predikcija \
  --auto-scaling-group-name moja-grupa --policy-type
PredictiveScaling \
  --predictive-scaling-configuration
file://config.json
```

`config.json`

- Ovo je isti `config.json` baš kao i onaj koji smo koristili za upravljanje vođeno metom
- Zašto? Zato što nas ovde samo zanima da definišemo metriku, ništa više.

`config.json`

```
{
  "TargetValue": 40.0,
  "PredefinedMetricSpecification":
    {
      "PredefinedMetricType":
        "ASGAverageCPUUtilization"
    }
  "Mode": "ForecastAndScale"
}
```

Upravljačka predikicija

- Sve što treba da uradimo da se postaramo da radi upravljačka predikcija jeste da modifikujemo `config.json` tako da ubacimo da je vrednost ključa “Mode” ravna “ForecastAndScale”
- Sva ostala konfiguracija je identična

Dozvoljeno probijanje granica

- Moguće je dozvoliti upravljačkoj predikciji da probije maksimalni broj instanci grupe
- To služi da opsluži situaciju kada mehanizam detektuje rastući trend upotrebe koji je veći nego što su ljudski planeri očekivali, a ne sme se dozvoliti da sistem nema odziv.
- Kada je takva situacija, veštačka inteligencija koja je iza ovog sistema može biti ovlašćena da prekrši pravila

`config.json`

```
{
    "MetricSpecifications": [
        {
            "TargetValue": 70,
            "PredefinedMetricPairSpecification": {
                "PredefinedMetricType":
        "ASGCPUUtilization"
            }
        }
    ],
    "MaxCapacityBreachBehavior":
        "IncreaseMaxCapacity",
    "MaxCapacityBuffer": 10
}
```

Proboj kapaciteta

- Prva opcija, `IncreaseMaxCapacity` dozvoljava sistemu za prediktivno upravljanje da privremeno poveća maksimalni kapacitet na onoliko koliko je predviđeno da će biti potrebno.
- Druga opcija, `MaxCapacityBuffer` dozvoljava sistemu da alocira nešto više resursa od predviđenih kao bezbednosni bafer
- U slučaju koji je prikazan, ovaj bezbednosni bafer je ravan 10% predviđenog kapaciteta
- Dakle ako je maksimum 30, a predviđeno je 40, moći će biti alocirano najviše 44.

Rizik

- Treba se strašno čuvati nekontrolisanog automatskog upravljanja
- Skaliranje nije samo tehnička nego i ekonomska odluka
- Veštačko inteligentni i automatizovani sistemi nemaju ‘zdravu pamet’: oni prate svoja pravila i modele, i mogu ih veselo pratiti preko ivice litice bez ikakvog problema.
- U jednom trenutku, primerak “The Making of a Fly” P. A. Lorenca je bila izlistana na prodaju na Amazonu za 23 698 655 dolara i 93 centi.
- Što?!

Balansiranje opterećenja

Balans za horizontalno skaliranje

Mehanizam balansiranja opterećenja

- Jako čest pristup horizontalnom skaliranju jeste da se nekakav broj zahteva rasporedi po više obrađivača
- Mehanizam odgovoran za ovu proceduru se zove ‘balanser opterećenja’ ili na engleskom “Load Balancer”
- Posao balansera je da se postara da svi zahtevi budu poslati na obradu na takav način da se mehanizimi horizontalnog skaliranja jednako opterete

Lokacija balansera opterećenja

- Balanser opterećenja može biti lociran, fizički, kao deo drugih mehanizama
- Može biti balansiran na samom mrežnom switch uređaju
- Može biti poseban dodatak na mrežnu infrastrukturu, kao odvojen uređaj
- Mehanizam operativnog sistema koji radi sličnu stvar, ali u okviru jednog sistema (tipično se raspoređivanje onda vrši na kontejnere)
- Servisni agent: nešto što pruža sam sistem za CC

Dodatne funkcije balansera opterećenja

- Asimetrična distribucija
- Određivanje prioritetske obrade
- Distribucija svesna sadržaja

Asimterična distribucija

- Najprostiji pristup jeste kada svi čvorovi za obradu imaju iste mogućnosti
- Ponekad ovo nije slučaj
- Tada, ‘fer’ distribucija nije ono što želimo
- Dobri balanseri mogu da to uzmu u obzir

Prioritetska obrada

- Nisu svi zahtevi nužno jednako važni
- Balansiranje može da u sebi uvrsti i QoS mehanizme koji se staraju da određeni zahtevi budu poslati na obradu ranije
- Ne samo to: prioritet može da utiče i na mnoge druge faktore kao što je, recimo, ko obrađuje nešto ili koji zahtevi mogu da se odbace (ako su, recimo, zahtevi sa kratkim periodom relevancije, tj. u pitanju je periodičan zahtev za novim informacijama)

Distribucija svesna sadržaja

- Ako je zahtev podložan dubljoj inspekciji može da se usmeri ka različitim tačkama u okviru sistema u zavisnosti od toga šta se nalazi unutar zahteva
- Ovo je način da se aktiviraju oni servisi obrade koji su u stanju da izvrše baš ono što zahtev traži i može da bude ključan element u slabo povezanim arhitekturama aplikacije

Primer balansiranja: `Nginx`

- `Nginx` je softver koji je primarno web server
- Zbog toga što je neobično performantan, `nginx` se često koristi kao softver-baziran balanser
- U ovoj arhitekturi, umesto hardverskog uređaja za balansiranje, spoljni saobraćaj dolazi do jednog računara na kome se izvršava `nginx` koji onda zahteve koje dobija šalje dalje, u skladu sa nekim pravilima.

Izvori

Primeri za `nginx` konfiguraciju su povučeni iz zvanične dokumentacije dostupne na **docs.nginx.com/nginx/admin-guide/**

`Nginx` **konfiguracija za balansiranje**

```
http {
    upstream backend {
        server backend1.example.com weight=5;
        server backend2.example.com;
        server 192.0.0.1 backup;
    }
}
```

Konfiguracija za balansiranje

- Ovo predstavlja kolekciju servera definisanih u okviru grupe ‘backend’
- Primetite da osim što server ima hostname ili IP ima i druge parametre
- `weight` služi da se radi asimetrična distribucija zahteva tako što se nekom odredištu daje veća ‘težina’ u distribuciji
- `backup` direktiva znači da dati server ima težinu 0 osim ako ostali serveri nisu isključeni

Konfiguracija za balansiranje

```
server {
    location / {
        proxy_pass http://backend;
    }
}
```

Konfiguracija za balansiranje

- Ovo definiše novi server koji sve zahteve koji stižu na korenski URI šalje dalje koristeći ‘backend’ grupu
- Ovo znači, naravno, da bi ovaj server mogao i da šalje zahteve različitim grupama u zavisnosti od putanje

Metode podele

- `Nginx` podržava veliki broj mehanizama za balansiranje:
  - Round robin
  - Nanjmanje veza
  - IP heširanje
  - Opšti heš
  - Najmanje vremena
  - Nasumično

Round robin

```
upstream backend {
    # Ovde ide specifikacija tipa podele, ali za round
robin ne ide ništa
    server backend1.example.com;
    server backend2.example.com;
}
```

Najmanje veza

```
upstream backend {
    least_conn;
    server backend1.example.com;
    server backend2.example.com;
}
```

Najmanje veza

- Ovo jednostavno raspoređuje zahteve onom serveru ka kome trenutno ima najmanji broj aktivnih veza (uzimajući težine servera u obzir, naravno)

IP heširanje

```
upstream backend {
    ip_hash;
    server backend1.example.com;
    server backend2.example.com;
}
```

IP heširanje

- Ovde se rasponi heš vrednosti distribuiraju između servera tako da je svaki odgovoran za neki deo ukupnog heš prostora
- Ovo se najlakše može raditi modulo aritmetikom
- Zatim se prva tri okteta dolazeće IPv4 adrese, odn. cela IPv6 adresa hešraju i na osnovu toga se određuje koji server će obraditi zahtev

IP Heširanje

- Uzimajući u pretpostavku da zahtevi dolaze uniformno sa različitih adresa, ovo se stara da distribucija bude ‘fer’
- Takođe, kao bonus, isto ishodište zahteva će uvek dobiti isti server, što je jasan bonus

Opšti heš

```
upstream backend {
    hash $request_uri consistent;
    server backend1.example.com;
    server backend2.example.com;
}
```

Opšti heš

- Ovde je sve isto kao IP heš, ali se hešira nešto drugo (recimo URI zahteva)
- `Consistent` znači da se koristi Ketama mehanizam koji omogućava da, kada serveri padnu, heš se adaptira
- Zašto bi ovo hteli?

Najmanje vremena

```
upstream backend {
    least_time header;
    server backend1.example.com;
    server backend2.example.com;
}
```

Najmanje vremena

- Najmanje vremena je tehnika koja kao faktor distribucije uzima server sa najmanje veza *i* najkraćim kašnjenjem.
- Kašnjenje se može meriti na nekoliko načina:
  - `header` - do dolaska prvog bajta zaglavlja odgovora
  - `last_byte` - do dolaska zadnjeg bajta punog odgovora
  - `last_byte inflight` - do punog odgovora, uzimajući u obzir nekompletne zahteve

Nasumično

```
upstream backend {
    random two least_time=last_byte;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;
    server backend4.example.com;
}
```

Nasumično

- Random direktiva određuje server nasumično
- Ovo je obično suviše haotično za stvarnu primenu, tako da se često koristi `two` pod-direktiva koja garantuje nasumičan odabir dva kandidata
- Između ta dva kandidata se bira jedan koji će primiti zahtev

Nasumično

- Odabir između dva kandidata može da bude po tri kriterijuma
  - `least_conn`
  - `least_time=header`
  - `least_time=last_byte`

Nasumično

- Nasumična distribucija, čak i uz modifikaciju `two` direktive je nepraktična za običnu primenu
- Koristi se u distribuiranim situacijama u kojima više balansera šalje zahteve istom skupu back-end čvorova.
- Ako imamo bolji uvid u situaciju sa pozicije centralnog sistema za balansiranje opterećenja, uvek ima boljih tehnika od ‘nasumično’

Load Balancing primer 2 - AWS

- Amazon zove ovo ‘ELB’ odnosno ‘Elastic Load Balancing’ i nudi ovu uslugu u nekoliko različitih *nivoa*
- Ovaj pristup sa nivoima ilustruje kompleksnu varijaciju koja postoji u tehnologijama balansiranja
- Ilustracije iz ove sekcije su povučene direktno sa AWS dokumentacije

Nivoi balansiranja u ELB

- Gateway odn. Ruterski
- Network odn. Mrežni
- Application Load Balancer odn. Aplikativni

Gateway nivo

- Ideja iza gateway nivoa balansiranja jeste balansiranje između individualnih IP-ova
- Ono što se distribuira su *paketi*
- Efektivno, ovaj mehanizam stoji na mrežnim rutama između izvora zahteva i odredištima zahteva i nevidljivo menja putanje

Gateway arhitektura

Mrežni nivo pristupa

- Mrežni nivo pristupa je pozicioniran jedan nivo više u OSI hijerarhiji
- Modifikacije se prave na nivou TCP/UDP veza
- Ono što se rutira su sada *veze* odnosno TCP segmenti i UDP datagrami
- Osim što podržava usmeravanje na tom nivou, može da se poveže i sa balansiranjem na nivou aplikacije

Mrežna arhitektura

Aplikativni nivo

- Aplikativni nivo balansira opterećenjem na način koji je svestan aplikacija koje se izvršavaju nad sistemom
- On je, dakle, na sedmom OSI sloju
- Podržava ulaze koji su HTTP/S, WebSocket, gRPC i usmerava te na sve od individualnih adresa do grupa za skaliranje
- Zbog toga što je svestan strukture aplikacije, tokovi saboraćaja se mogu kontrolisati na mnogo fleksibilniji način

Arhitektura aplikativnog nivoa

Primer generisanja konfiguracije za balanser aplikativnog nivoa

```
aws elbv2 create-load-balancer --name balanser0  \
--subnets $SUBNET0 $SUBNET1 \
--security-groups $SG0 --ip-address-type dualstack
```

Primer generisanja konfiguracije za balanser aplikativnog nivoa

- elbv2 je samo odabir grupe funkcija koja se koristi, to je normalno
- Primetite da moramo definisati više javnih mreža; u skladu sa dokumentacijom one su javne i pripadaju različitim zonama dostupnosti
- `dualstack` opcija nije neophodna, ona samo omogućava da se koristi i IPv4 i IPv6 istovremeno
- Ova komanda vraća ARN koji identifikuje 'balanser0' imaće oblik tipa `arn:aws:elasticloadbalancing:us-east-2:123456789012:loadbalancer/app/balanser0/123456789012345`
- Kada treba da je pomenemo, mi kažemo: $LBARN

Digresija: 'zone dostupnosti'

- Zone dostupnosti je AWS termin za relativno nezavisne grupacije serverskih resursa koje postoje u okviru nekog geografskog regiona.
- Služe da budu međusobno jako dobro povezane dok su istovremeno maksimalno nezavisne: ideja je ako eksplodira transformator ili tako nešto u jednoj, da druge mogu da nastave da rade nesmetano.
- Upotreba ovih zona je, predividivo, neophodna za istinski robusno korišćenje oblaka
- Amazon nije jedini koji ovo nudi, naravno.

Kreiranje grupe meta

```
aws elbv2 create-target-group --name moje-mete --
protocol HTTP --port 80 \
--vpc-id $VPC0 --ip-address-type ipv4
```

Kreiranje grupe meta

- Ovde kreiramo mete na koje će biti usmereni zahtevi
- Primetite da su ovo zahtevi koji idu na port *80*
- VPC0 je virtuelna mreža u kojoj se nalaze EC2 instance koje obrađuju zahteve
- Ovo će vratiti ARN koji zovemo $TARN

Kreiranje grupe meta

```
aws elbv2 register-targets --target-group-arn $TARN  \
--targets Id=$INSTANCE0 Id=$INSTANCE1
```

Kreiranje grupe meta

- Ovim se registruju odredišta za load balancing
- Ovde eksplicitno imenujemo (virtuelne) računare
- Mogli smo da imenujemo (preko odgovarajućeg ARN-a) i Lambda instance, ili da stavimo IP adrese direktno ili čak ARN-ove drugih balansera
- Da bi ovo integrisali sa auto-skaliranjem od ranije moramo da specificiramo povezivanje sa load balanserom tokom pravljenja grupe za autoskaliranje

Povezivanje ELB-a i grupe meta

```
aws elbv2 create-listener --load-balancer-arn $LBARN \
--protocol HTTP --port 80  \
--default-actions Type=forward,TargetGroupArn=$TARN
```

HTTPS podrška

- Bezbednost je, naravno, ključna stvar kada se priča o podršci u oblaku
- Ovo stvara zanimljiv problem: ako imamo, potencijalno, 150 instanci koje uslužuju zahteve i ako želimo da koristimo HTTPS podršku sa klijentima (a danas, nema opravdanja da to ne činimo) gde se onda nalazi serverski sertifikat?
- Valja napomenuti da su serverski sertifikati *privatni ključevi* kojima potpisujemo naš saobraćaj napolje i koji spoljašnji klijenti koriste da verifikuju naš identitet

Menadžment ključevima

- Da li onda ovo znači da naše unapred-spremne konfiguracije moraju da imaju, kao deo svega toga, privatan ključ u nekakvom fajlu?
- Ovo je bezbednosna noćna mora: odjednom je bukvalno najosetljiviji podatak koji imamo dostupan jako široko
- Još gore, u formi je **fajla** koji može da se očita, kopira, eksfiltrira...
- Nikada, ikada ne želimo da je naš ključ u obliku da može da se jednostavno pročita (Šta je alternativa?)

HTTPS offloading

- Možda ste ranije videli na dijagramima odavde ovu frazu
- Ideja je da, kao deo vašeg ELB sistema takođe se postarate da odradite sve što treba za HTTPS transparentno
- Vaše instance onda mogu da žive u lepom, mirnom HTTP svetu (zato što je sav interni saobraćaj enkriptovan)
- A vi pristupate osetljivim kriptografskim podacima na jednom centralizovanom mestu

Podešavanje HTTPS

```
aws elbv2 create-listener --load-balancer-arn $LBARN \
--protocol HTTPS --port 443  \
--certificates CertificateArn=$CARN \
--default-actions Type=forward,TargetGroupArn=$TARN
```

Odakle vam CARN?

- Amazon ima mehanizam koji automatizovano generiše, upravlja sa, i obnavlja sertifikate (i povezane privatne ključeve, naravno) za vaše sajtove.
- Možete takođe koristiti ovu infrastrukturu da čuvate ključeve koje vam je neko drugi dao
- Čak možete da podignete vaš sopstveni rootCA i onda upravljate sertifikatima koje sami izdajete uz upotrebu CloudHSM mehanizma da bi ste čuvali vaš rootCA.

Menadžment bezbednošću sa harverskom potporom

‘Skupa’ bezbednost

Terminologija

- Menadžment bezbednošću sa hardverskom potporom je generički termin za familiju tehnologija gde se poseban bezbednosni hardver koristi za posebne funkcije bezbednosti.
- Ovo se razlikuje od toga što je hardver neophodan da bi računar imao *bilo kakve* mehanizme bezbednosti, budući da sve to zavisi od harvera dizajniranog za tu svrhu.
- Ovde govorimo o hardveru koji pruža funkcije bezbednosti koje su nezavisne od i na neki način ‘iznad’ npr. procesora.

Lokus kontrole

- Još jedna ključna razlika je u lokusu kontrole: ko ima kontrolu nad uređajem i kodom koji se izvršava na njemu.
- Tehnike klasične bezbednosti već razdvajaju kod: po virtuelnim mašinama ili kao privilegovan i neprivilegovan kod, vlasnik hardvera ima apsolutnu kontrolu nad tim šta se izvršava gde.
- Kod hardverskih bezbednosnih komponenti o kojima hoćemo da pričamo, u pitanju je *proizvođač* koji ima lokus kontrole ili, često, ne čak ni proizvođač no lokus pripada samom uređaju koji ima gvozdena pravila o tome šta sme kako da se izvršava i ko sme šta da vidi i to što mi imamo hardver nam ne dozvoljava da biramo šta će da se izvršava.

Problem predpostavke poverenja

- Ovo znači da, ako koristimo ovakve sisteme (a ako koristimo CC, koristimo ih, ovako ili onako) ti uređaji izvršavaju kod o kome mi ništa ne znamo, i nad kojim nemamo kontrolu.
- Najosetljivije operacije našeg računara onda postanu stvar neprozirne crne kutije.
- Ovo je potencijalno veliki problem iz tri glavna razloga:
  - Institucijalna uverenost
  - Akrecija poverenja
  - SPOF

Uverenost

- Često je slučaj da ne samo što moramo da učinimo računarski sistem bezbednim na način da smo mi zadovoljni ili da su naši klijenti zadovoljni, no možda moramo da dokažemo bezbednost nekoj trećoj strani.
- Ta treća strana može (i često ima) veoma specifične standarde koji određuju šta je to, tačno, 'bezbedno'
- Veliki deo ovog 'uveravanja' jeste da postoji ogromna transparentnost oko toga kako sistem radi: gotovo uvek to znači da će kod biti deljen da bi mogla da se radi nezavisna provera bezbednosti.
- Ako ključni elementi bezbednosti žive u crnoj kutiji, uveravanje će ići mnogo teže.

Akrecija poverenja

- Ako proizvođač hardverskog bezbednosnog sistema ima kontrolu nad njim (što nije uvek slučaj) i ako ta kontrola se odražava u posedovanju određenih tajni, kriptografskih i drugačijih, onda je nužno da se veruje da proizvođač:
- Nikad neće zloupotrebiti te podatke
- Nikada neće prodati te podatke
- Nikad neće biti pravnim putem prinuđen da oda te podatke
- Nikad neće, nehatom, 'procureti' podatke
- Ovo je puno poverenja da se stavi u jedan entitet

SPOF

- Kratko rečeno: ako je sva bezbednost podređena hardverskom sistemu i taj jedan sistem je provaljen, onda je ceo ostatak bezbednosnog sistema i sistema koji vrši upravljanjem rizika u slučaju bezbednosnog proboja nebitan.
- Za nekakve sisteme ovo je sasvim neprihvatljivo.

Prednosti

- Ako imamo sve ove probleme, što koristimo ovo?
- Zato što ima stvari koje ne možemo da uradimo bez njih.
- U slučaju jednog računara predstavljaju jedinu moguću odbranu od 'zla sobarica' napada
- Anatomija direktnog pristupa.

Prednosti u CC okruženju

- U CC okruženju to da neka treća strana ima pristup našem hardveru je data stvar
- Plus, jednako ‘data’ je i stvar da će naš hardver da dele drugi korisnici, a ma koji od njih mogu biti maliciozni.
- Ponekad hardverska bezbednost je sve što možemo uraditi.

Šta to tačno radi hardverkski uređaj bezbednosti?

- Glavna funkcija ovakvih uređaja jeste da se stvori nekakva particija računarskog sistema, virtuelizovanog ili ne, koja igra po sopstvenim pravilima.
- Zamislimo, recimo, da imamo ‘particiju’ koja čuva naše ključeve u regionu memorije kome se ne može čak ni **teoretski** pristupiti.
- Onda ne moramo da se plašimo da će neko ukrasti ključ: ne može a da fizički ne iščupa komad hardvera iz našeg računarskog sistema, a to bi primetili.
- Kako bi izgledao hardverski uređaj bezbednosti koji nas štiti od ‘zle sobarice?’

Kriptografski moduli

- Centralna ideja za kripto-modul dolazi iz vrlo ozbiljnog problema čuvanja nekakve kriptografske tajne
- Ako je tajna kompromitovana, naročito ako o tome nije svestan vlasnik tajne, sva buduća upotreba kripto-sistema koji se oslanja na tu tajnu je kompromitovana.
- Problem je što na konvencionalnom računarskom sistemu tok podataka je **namerno** maksimalno lak: ako možemo da nešto šifrujemo, onda možemo da pristupimo ključu da bi ga čitali, a onda moramo da možemo i da ga ukrademo.

Kriptografski moduli

- Ako implementiramo, onda, kriptografiju kroz običan softver tajne koje koriste moraju biti negde na disku i negde u memoriji i to znači da ovako ili onako mogu da budu ekstrahovane.
- Ovo je veoma često neprihvatljivo
- Kao rezultat kriptografkse operacije se često implementiraju u posebnim integrisanim kolima
- Ta integrisana kola (najčešće nekakva kartica ili tako što) se zovu ‘kriptografski modul’ ili ‘hardverski bezbednosni modul’
- Ozbiljni provajderi cloud usluga će vam iznajmiti ekskluzivan pristup ovakvim uređajima ako to želite.

Što kripto modul?

- Izolacija algoritma
- Izolacija tajni
- Performanse
- Otpornost na subverziju
- Evidentnost subverzije

Izolacija algoritma

- Vrlo je teško napisati dobru implementaciju kripto-sistema kao što smo dobro i sami saznali
- Imati implementaciju koja je već prošla rigoroznu validaciju je dobra stvar, ali naravno, i dobra biblioteka čini isto
- Ali ima i dublja prednost u izolaciji

Izolacija algoritma

- Naš kriptografski kod, bilo da smo ga sami pisali ili da koristimo kvalitetnu biblioteku se izvršava na istom mestu kao i sav drugi kod
- Sav taj drugi kod, u modernom programerskom svetu, verovatno ima čitavu ‘šumu’ paketa i modula od kojih zavisi
- Ti paketi i moduli, sa druge strane, zavise od druge šume paketa i modula
- Kontrola ovoga, naročito ako uzmete to kako se menjaju verzije je *jako* teška.

Izolacija algoritma

- Sve ove zavisnosti pružaju kataklizmično ogromnu površinu za napade ili kroz ranjivosti i greške u samim tim dodatnim modulima koje možda dozvoljavaju nekakavu lukavu tehniku pristupa sa strane memoriji koju koristimo za kriptografiju
- Stvar je još gora: maliciozni paketi omogućavaju da se lansira napad iz *unutrašnjosti* našeg koda: odatle je pristup kriptografskim tajnama direktan i trenutan.

`flatmap-stream`

- U novembru 2018 je oktriveno da je napadač pruzeo ulogu održavanja na FOSS paketu `flatmap-stream` koji je u okviru `npm` sistema bio dependency jako puno korišćene `event-stream` biblioteke.
- Kada su preuzeli kontrolu, napadač je objavio novu verziju, 3.3.6 koja je bila ista kao i prethodna samo što je još i krala kripto-valutu (*naravno* da jeste)
- Napad je bio potpuno socijalan: napadač je ubedio ljude da će biti odgovoran vlasnik tog projekta

Post-hoc trojanski konj

- Sav kod u okviru jedne aplikacije se izvršava sa istim privilegijama i deli isti memorijski prostor
- To znači da ako neko uspe da infiltrira maliciozni dependency u sistem već ima *potpunu kontrolu.*
- Nema potrebe ništa drugo da radi, već je vlasnik vašeg sistema.

Post-hoc trojanski konj napad

1. Napisati korisnu biblioteku na nekom velikom javnom repozitorijumu: npm, pypi, NuGet... ili preuzeti postojeću kroz socijalni inženjering.
2. Sačekati da ta biblioteka postane široko korišćena, idealno kao dependency nečega još popularnijeg.
3. Modifikovati biblioteku tako da radi nešto maliciozno

Post-hoc trojanski konj napad

- Tačno se ovo i desilo popularnoj bibloteci `electron-native-notify`
- A ne samo i njoj nego i još par drugih NPM paketa

...par...

- Axioss
- Axios-http
- Body-parse-xml
- Sparkies
- Js-regular
- File-logging
- Mysql-koa
- Import-mysql
- Mogodb
- Mogobd
- Mogoose
- Mogodb-core
- Node-ftp
- Serializes
- Serilize
- Kao-body-parse
- Node-spdy

Joj

- Ovo je samo jedan prolaz kroz arhive koji je već zastareo
- *Ekstremna* pažnja je neophodna da se obezbedi programersko ogruženje

Izolacija tajni

- Već pomenuto: sve što je u memoriji je, teoretski, čitljivo
- Čak i ako je nešto u memoriji drugog procesa ili druge virtuelne mašine, postoji *nekakva* sekvenca koraka koja tu memoriju iščitava: tu je, na istim je adresnim linijama.
- HSM-ovi mogu da čuvaju tajne u posebnoj memoriji koju samo oni mogu da čitaju
- Alternativa je još ozbiljnija: memorija koja se *fizički* ne može kopirati

Memorija koja se ne može kopirati

- Klasičan HSM pristup jeste da se jednostavno napravi obična memorija koju samo sme da čita specijalizovan čip kome je naređeno da nikome ne oda sadržaj i koja je inženjerisana tako da se samouništi u slučaju da neko pokuša da manipuliše uređajem
- Mučna ranija istorija je pokazala da nismo baš... 100% kada je u pitanju pravljenje ovakvih uređaja.
- Alternativa je nešto što, štagod da se desilo, *ne može* da se kopira.

Fizički neklonabilne funkcije

- Fizički neklonabilne funkcije uzimaju nesavršenosti i pretvaraju ih u prednost
- Svako integrisano kolo ima malecne varijacije u ponašanju
- Te malecne varijacije su manje-više jedinstvene i nasumične (proističu iz defekta u proizvodnji) i ako bi pokušali da ih kopiramo, uništili bi tokom analize baš te tanane varijacije koje te razlike stvaraju.
- PUF uzima te nesavršenosti i pojačava ih do nivoa da ih je lako iščitati.

Primer PUF-a

Primer PUF-a

Slika iz: Mostafa, A., Lee, S. J., & Peker, Y. K. (2020). Physical Unclonable Function and Hashing Are All You Need to Mutually Authenticate IoT Devices. Sensors (Basel, Switzerland), 20(16), 4361. https://doi.org/10.3390/s20164361 Creative Commons Attribution License