18th International Symposium INFOTEH-JAHORINA, 20-22 March 2019

Modelovanje sistema za upravljanje tokovima poruka primjenom Apache Kafka

Nebojša Kuduz, Ozren.Ćalić, Slaven Popović
Telekomunikacije Republike Srpske a.d., Banja Luka
Nebojsa.Kuduz@mtel.ba, Ozren.Calic@mtel.ba, Slaven.Popovic@mtel.ba

Sadržaj - **U radu je predložena realizacija brokera VAS poruka za procesiranje velikog broja SMS poruka upotrebom Apache Kafka tehnologije. Riječ je o optimizovanom rješenju za razmjenu poruka koje se pokreće na OpenStack oblaku. Sistem koristi Apache Kafka koji u svom algoritmu objedinjuje obradu tokova i klastering i brine o replikaciji, redundansi, skaliranju node-ova unutar Kafka klustera. Primjenom Kafka klastera povećana je horizontalna skalabilnost i procesni kapacitet sistema za obradu poruka. Sistem je dizajniran da služi za brzu, pouzdanu i dvosmjernu razmjenu VAS poruka između VASP korisnika i provajdera telekomunikacionih usluga. Primjenom Kafke značajno je povećan kapacitet obrade poruka u odnosu na primjenu AMQ brokera. Predloženi model treba da prevaziđe troškove licenciranja sistema jer je u potpunosti zasnovan na stabilnim opensource rješenjima. Sistem u fokusu ima Apache Kafka klaster na virtualizovanoj platformi što omogućuje paralelnu obradu podataka i značajno bolje performanse u odnosu na referentni sistem za obradu poruka koristi JGroups i Logica SMPP library, ActiveMQ broker poruka i Oracle RAC. Upotreba Kafka klastera i pravilno konfigurisanje brokera garantuje isporuku poruka odgovarajućim korisnicima. Koristili smo Kafka Streams API klijentsku biblioteka za izgradnju aplikacija gdje se i ulazni i izlazni podaci pohranjuju u Kafkin klaster. Kafka Streams API nam je omogućio kombinovanje jednostavnog kodiranja implementacijom standardnih Java i Scala aplikacija na strani klijenta i prednosti tehnologije Kafka klastera na strani servera.**

Ključne riječi- Kafka, Kafka Streems API, Zookeeper

I. Uvod

VAS (Value Added Service) sistem za razmjenu poruka omogućuje sinhronu ili asinhronu razmjenu informacija između različitih provajdera VAS usluga i telekomunikacionih operatera. Razmjena informacije se obavlja putem brokera poruka koji je posrednik odnosno pomoćni programski modul koji prevodi poruku iz formalnog protokola za slanje poruka pošiljaoca u formalni protokol za prijem poruka na strani primaoca. U telekomunikacionoj arhitekturi mreže brokeri i sistem za razmjenu poruka su pozicionirani između VAS provajdera i centara za razmjenu poruka odnosno SMSC/MMSC. VAS aplikacije u telekomunikacijskim mrežama su softverske aplikacije koje komuniciraju i razmjenjuju formalno definisane SMS/MMS/WAP poruke putem platformi za razmjenu poruka.

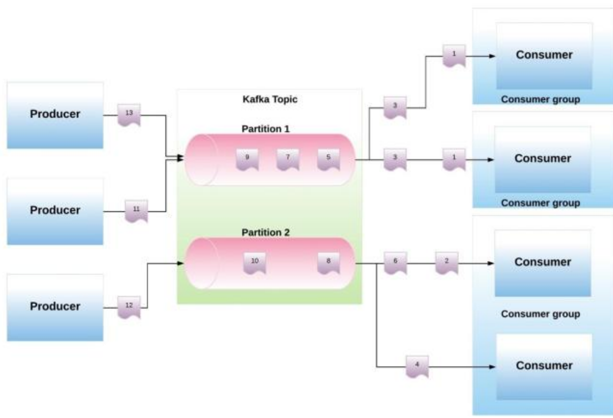
Arhitekture postojećih brokera koji se koriste u telekomunikacionim sistemima za razmjenu i obradu velikog broja VAS poruka uglavnom imaju ograničene mogućnosti procesiranja, replikacije, redundanse i horizontalnog skaliranja. To su uglavnom zatvorena rješenja zasnovana na prevaziđenim softverskim konceptima, koriste komercijalne licence i komplikovana su za održavane. Sva rješenja ovoga tipa treba da imaju visoku dostupnost, ali su procesni kapaciteti ograničeni jer su uglavnom pokrenuta na neskalabilnom softveru i hardveru koji ne omogućuje paralelno procesiranje ulaznih i izlaznih tokova podataka, a pohrana podataka je uglavnom u relacionim bazama koje unose dodatno kašnjenje u tok procesiranja poruka.

VAS sistem razmjene poruka predstavlja SMS *gateway* koji prima poruke od korisnika, rutira ih i šalje odgovarajućem SMS servis provajderu koji ih dostavlja krajnjem korisniku. On prima izvještaje o isporuci od pružoca usluge i šalje ih nazad korisniku koji je poslao originalnu poruku. Poruke se mogu slati i u suprotnom pravcu - od krajnjeg korisnika do klijenta Primarni protokol koji koristimo za komunikaciju između korisnika i VAS sistema, kao i između VAS sistema i provajdera SMS usluga je SMPP [1].

Sistem za obradu velikog broja poruka mora da bude modelovan na takav način da ima visoku dostupnost i da može da podnese velika opterećenja, pa horizontalno skaliranje mora biti podržano. To je sve uticalo na izbor Apache Kafka u modelovanju sistema za procesiranje ulaznih i izlaznih tokova podataka i velikog broja poruka.

II. Model distribuiranog sistema za obradu poruka

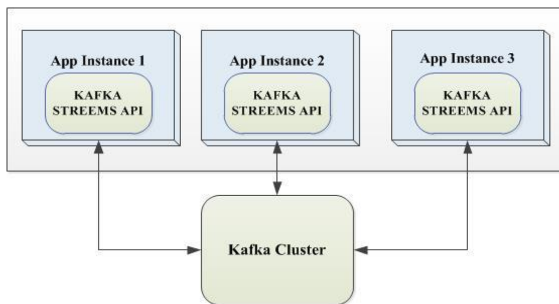
Kafka je prvobitno dizajnirana od strane LinkedIn-a, napisana je u Scali, a sada je pod pokroviteljstvom Apache zajednice. Pojednostavljeno Kafku možemo smatrati ogromnim redom u koji možemo pohraniti mnogo podataka koje treba obraditi, a da ne moramo obavezno obraćati pažnju kada će se obrada izvršiti. Na jednoj strani imamo programe odnosno procese Proizvođače koji šalju poruke Kafki u određene Teme. Na drugoj strani imamo programe odnosno procese Potrošače ili Pretplatnike na Teme koji čitaju poruke i po potrebi ih dalje obrađuju. Potrošači pripadaju barem jednoj grupi potrošača, koja se povezuje s nekom Temom. Svaki Potrošač unutar grupe mapira se na jednu ili više particija teme. Kafka osigurava da poruku čita samo jedan Potrošač u grupi i da sve poruke koje pripadaju istoj Temi budu dostavljene svim registrovanim korisnicima grupe Sl.1.



Slika 1. Kafka Tema, Proizvođač i Potrošač

Svaki Potrošač čita iz particije i prati ofset odnosno pomak. Ako potrošač koji pripada određenoj grupi potrošača ispadne, Kafka dodjeljuje particiju nekom od drugih potrošača. Slično tome, kada se novi potrošač pridruži grupi, on se sinhronizuje i uravnotežuje asocijaciju particija sa ostalim dostupnim potrošačima. Moguće je da se više grupa potrošača pretplate na istu temu. Na primjer, grupa potrošača može primati poruke za obradu u realnom vremenu putem nekog sistema za obradu podataka, a druga grupa potrošača može primati poruke iz iste Teme radi njihovog smještanja u bazu podataka.

Kafka je u suštini masivno skalabilni red za poruke organizovan na principu Proizvođač/Pretplatnik, modelovan kao distribuirani transakcioni log što ga čini vrlo korisnim u infrastrukturama telekomunikacionih kompanija [2],[3]. Kafka se koristi u sprezi sa alatom ZooKeeper za upravljanje instancama i za praćenje performansi u Kafka klastera [4]. Za pokretanje Kafka klastera prvo je potrebno pokrenuti Zookeeper Sl.2.



Slika 2. Kafka ZooKeeper distribiranog sistem sa tri Brokera

ZooKeeper je distribuirani, hijerarhijski sistem datoteka koje su reaspoređene poput fajlova i direktorijuma u tradicionalnom fajl sistemu. ZooKeeper postiže visoku dostupnost pokretanjem više ZooKeeper servera koji formiraju klaster. Svaki server drži u memoriji kopiju distribuiranog sistema datoteka kako bi opslužio zahtjeve Potrošača. Svaki Kafka broker koordinira sa drugim Kafka brokerima koristeći ZooKeeper.

Kafka čuva poruke u kategorijama koje se nazivaju Teme. Teme se sastoje od jedne ili više Particija. Svaka Particija je uređena, nepromjenjiva sekvenca poruka kojoj se stalno dodaju novi zapisi. Poruke se ne brišu nakon isporuke, već nakon konfigurisanog vremenskog perioda. Svaka Kafka instanca koja pripada klasteru se zove Broker. Broker je instance Kafka servera i njegova primarna uloga je da prima poruke od Proizvođača, dodjeljuje ofset i smješta poruke na disk.

Osnovna razlika između Kafka brokera i drugih brokera poruka je da klijenti nikada neće automatski primati poruke, već moraju eksplicitno tražiti poruku kada su spremni za rukovanje. Sa odgovarajućim hardverom, svaki broker može lako da obradi hiljade particija i milione poruka u sekundi. Više Brokera formira Kafka klaster. Jedan Broker može da obradi hiljade čitanja i zapisivanja u sekundi. ZooKeeper mora da ima mogućnost sigurnog izbora Brokera. Broker hostuje Teme i tematske particije, čak i kada Tema ima samo jednu particiju. Proizvođači šalju poruke Brokeru, nakon toga Broker ih pohrani na disk i markira ih jedinstvenom ofsetom. Na osnovu Teme, Particije i ofseta, Broker omogućuje Potrošačima da pristupe porukama. Brokeri unutar klastera međusobno razmjenjuju informacije direktno ili indirektno koristeći ZooKeeper.

Kafka klaster ima tačno jednog Brokera koji djeluje kao Kontrolor. Kontrolor je odgovoran za rukovođenje administrativnim operacijama, kao i dodjeljivanje particija drugim brokerima. Kontroler takođe prati stanje ostalih brokera. Kafka Broker je prilično jednostavan i ne brine mnogo o Potrošačima. Ta jednostavnost ga čini super brzim i nezahtjevnim u pogledu potrošnje resursa.

Slično kao i relacijske baze podataka, Kafka može pružiti trajni zapis svih transakcija koje se mogu reprodukovati da bi se oporavilo stanje sistema. Podaci su trajno pohranjeni u redoslijedu koji se može deterministički čitati. Particije u temi mogu se distribuirati na više brokera. Zahvaljujući distribuiranom dizajnu, Kafka osigurava redundantnost i visoku dostupnost podataka čak i kada se desi ispad nekog od nodova klastera. Particije se repliciraju preko Kafka klastera.. Faktor replikacije se podešava za svaku Temu. Kopije particija se distribuiraju između više brokera, ali samo jedan broker je Kontrolor što znači da sve što se piše i čita na tu particiju mora biti obrađeno preko tog brokera. Drugi brokeri se koriste za redundansu. Ako se broker Kontrolora sruši, Kafka će automatski izabrati drugog Kontrolora među preživjelim brokerima.

Poruke se čuvaju neko vrijeme i mogu se po potrebi konzumirati više puta preko ponovno postavljenih pokazivača. Postojanost poruka je vremenski ograničena i Potrošač je odgovoran da konzumira relevantne poruke prije nego ih Broker izbriše. Današnji Brokeri imaju približno isti ukupni kapacitet koji je ograničen na ~75 MB/s. Kafka pokazuje mnogo bolje performanse u obradi malih poruka (10-1024 bajta) u poređenju sa drugim brokerima i nudi horizontalno skaliranje pa je logičan izbor za modelovanje arhitekture sistema za procesiranje SMS poruka.

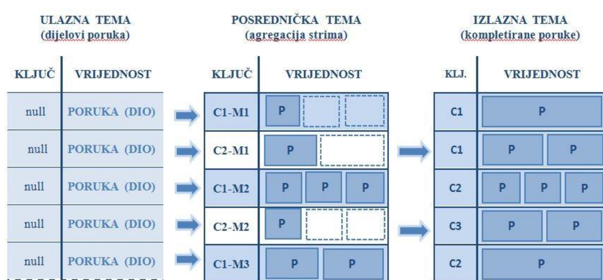
III. IMPLEMENTACAIJA BROKERA VAS SMS PORUKA

U implementaciji smo koristili Kafka Streams API koji omogućuje obradu tokova podataka u realnom vremenu [5-7]. Kafka Streams API može da agregira više tokova, spaja podatke iz više tokova, rješava probleme sa redoslijedom zapisa itd. Kafka Streams uzima neprekidne tokove zapisa iz ulaznih tema, vrši njihovu obradu, transformacije, agregacije i proizvodi jedan ili više izlaznih tokova.

Jedan problem sa SMS-om je da su poruke ograničene na 160 znakova u slučaju 7-bitnog kodiranja ili 70 znakova za UCS-2 kodiranje, veličine su 128 bajta da bi se uklopile u postojeće formate signalizacije. Način na koji SMS protokol prevazilazi ovo ograničenje je korištenje takozvanih dugih SMS poruka. Duge SMS poruke se šalju kao dijelovi konačne poruke koja se prikazuje krajnjem korisniku tek nakon što pristignu svi njeni dijelovi. SMPP protokol podržava ovu funkciju, tako da je bilo potrebno da i aplikacija podrži istu funkcionalnost. Kratke poruke su jednodijelne i logički nisu duge poruke, ali iz tehničke perspektive sve poruke se lakše tretiraju na isti način, tako da ne pravimo strukturalne razlike između pojedinačnih poruka i dugih poruka.

Aplikacija je potrebno da radi sa klasterom, sa više aktivnih brokera, a klijenti mogu biti povezani na bilo koji od brokera. Korisnici se povezuju na istu virtualnu IP adresu i učitavaju poruke, a balanser ravnomjerno distribuira veze na raspoložive brokere. Korisnik može imati više aktivnih veza istovremeno, a može i slati različite dijelove iste duge poruke putem različitih veza. Zbog toga je potrebno da agregiramo dijelove poruka koji dolaze od korisnika kroz jednu ili više veza uspostavljenih sa bilo kojim brokerom. I za agregiranje dijelova poruka takođe smo koristili Kafku koja ne zahtijeva direktnu replikaciju podataka između aplikacijskih nodova.

Imamo dvije teme, prvu za čuvanje djelova poruka, a druga služi za smještaj konačne poruke, odnosno poruke koje sadrže sve dijelove. Kada je poruka primljena od korisnika, ona sadrži podatke koje možemo koristiti za ispitivanje da li je to jednodjelna ili višedjelna poruka. Ako se radi o jednom dijelu, nova instanca konačne poruke (klasa "KonacnaPoruka") se kreira i upisuje se pomoću Proizvođača direktno na temu koja sadrži konačnu poruku Sl.3.



Slika 3. Način prenosa podataka

U suprotnom, ako je od korisnika primljena duga poruka, izdvajamo informacije kao što je identifikator poruke kojoj ovaj dio pripada, broj (indeks) dijela i ukupni broj dijelova koji sadrži konačna poruka. Zatim kreiramo instancu dijela poruke (klasa "PorukaDio"), popunjavamo je sa ovim informacijama i upisujemo u je u temu "DijeloviPoruke" koristeći Kafka Proizvođač.

Posao povezivanja sa klijentima je time završen i ne treba da brinemo kako će dijelovi poruke biti sastavljeni ili obrađeni. Sastavljanje poruke vrši KafkaStreams API.

Koraci koje je potrebno napraviti su: konfigurisati aplikaciju za obradu tokova, definisati topologiju aplikacije koristeći KStream/KTable operacije i pokrenuti aplikaciju.

Konfiguracija zahtijeva "ID aplikacije". Termin aplikacija se odnosi na skup procesa koji pokreću KafkaStreams topologiju putem aplikacijskih nodova. Ova osobina je vrlo slična group_id u konfiguraciji Kafka Potrošača i koristimo je da kontrolišemo način na koji višestruke instance KafkaStreams procesa pristupaju particijama Teme. Isto kao i kod grupa potrošača, više KafkaStream instanci sa istim ID-om aplikacije ne mogu istovremeno pročitati istu particiju Teme. Broj particija ograničava broj primjeraka aplikacije za *streaming* koje mogu istovremeno čitati iz Teme. Aplikaciji dajemo instrukciju kako da pronađe Kafka klaster tako što navodimo više brokerskih adresa odvojenih zarezom u obliku {host}:{port}. U našem slučaju imamo tri brokera.

Za serializaciju i deserializaciju ključeva i vrijednosti zapisa u Temi koristimo SerDes metode, kombinacija serializatora i deserializatora. Koristili smo SerDes metode koje je već obezbjedila Kafka biblioteka. Na pojedinim mjestima smo trebali transformirati ključ ili vrijednost ulaznih tema u neki drugi tip na izlazu, pa je potrebno modifikovati i kreirati posebne serdes metode.

U obradi poruka izabrali smo metodu "tačno jednom" kojom dajemo instrukcije KafkaStreams da želimo da samo jednom obradi svaku poruku iz ulazne Teme. Podrazumijevana vrijednost ove metode je *fault* što znači da će se poruke obrađivati u "barem jednom" načinu. U ovom režimu sve poruke će biti obrađene, ali neke poruke mogu biti obrađene dva puta u slučaju pada aplikacije noda.

Postoje tri različita Kafka Stream API-ja: DSL, Processor i KSQL. Mi smo odabrali Kafka Streams DSL (Domain Specific Language) [8]. KafkaStreams DSL pruža operacije za transformaciju, filtriranje, agregiranje, mapiranje itd.

Tema VaspDijeloviPoruke sadrži dijelove poruka koje potiču od konekcija različitih Pošiljaoca. Poruka je vrijednost zapisa u ovoj Temi, a ključ je null. Objekti za poruke imaju sve informacije koje su nam potrebne (ID korisnika, ID poruke, broj dijela poruke, ukupan broj dijelova). Koristeći *null* za ključ, postižemo punjenje particija u *round-robin* modu i na taj način ravnomjernije raspoređujemo podatke među Kafka brokerima i vršimo paralelnu obradu podataka. Broj particija u ovoj Temi je postavljen na 3, tako da možemo obraditi podatke sa 3 aplikacija istovremeno, jer imamo 3 instance aplikacije za obradu tokova podataka.

Prvo definišemo ulaznu temu iz koje se napaja aplikacija za obradu toka podataka. Metoda *builder.stream()* stvara izvor toka podataka. Zapisi u tom toku su isti kao u ulaznoj temi, ključ = *null*, vrijednost = dio instance poruke Sl. 4.

```
val builder = StreamsBuilder()
val ulazniTok = builder.stream<Long, PorukaDio>("PorukaDio")
ulazniTok
        .groupBy({ _, value ->
            "${value.posiljaocId}-${value.posiljaocPorukaId}"
        }, Serialized.with(Serdes.String(),
PorukaDioSerde()))
        .aggregate(
                KompletnaPorukaInitializer(),PorukaDioAgregator(),
                Materialized.with(Serdes.StringSerde(),
KompletnaPorukaSerde())
        )
        .toStream()
        .filter { _, value -> value.parts.size == value.totalPartsCount }
        .map { _, value -> KeyValue(null as Long?, value) }
        .to(
                "KompletnaPoruka",
                Produced.with(Serdes.Long(), KompletnaPorukaSerde())
        )
streams = KafkaStreams(builder.build(), streamConfig)
streams.start()
```

Slika 4. Obrada ulaznog toka podataka

Najvažniji dio je agregacija. Za agregiranje dijelova koji pripadaju istoj poruci, prvo grupišemo dijeove poruka sa složenim ključem koji kombinije *posiljaocId* i *posiljaocPorukaId*. *PosiljaocId* je id korisnika i neophodan je jer različiti pošiljaoci mogu imati iste ID-ove poruka. Kada imamo dijelove poruka grupisane sa ovim ključem, agregiramo te dijelove. Koristimo *aggregate()* metodu koja zahtijeva instance *org.apache.kafka.streams.kstream.Initializer* i *org.apache.kafka.streams.kstream.Aggregator* kao argumente. Inicijalizator se koristi za kreiranje praznog objekta rezultata agregacije, dok se Agregator koristi za dodavanje dijelova odgovarajućem objektu rezultata agregacije Sl. 5.

```
inner class PorukaDioAgregator : Aggregator<String, PorukaDio,
KompletnaPoruka> {
    override fun apply(msgId: String, part: PorukaDio, message:
KompletnaPoruka): KompletnaPoruka? {
        val updatePoruka = KompletnaPoruka(part.posiljaocId,
part.posiljaocPorukaId, part.partsCount)
        updatePoruka.parts.addAll(message.parts)
        updatePoruka.parts.add(part)
        return updatePoruka
    }
}
.toStream()
.filter { _, value -> value.parts.size == value.totalPartsCount}
.map { _, value -> KeyValue(null as Long?, value)}
.to(
        "KompletnaPoruka",
        Produced.with(Serdes.Long(), KompletnaPorukaSerde())
```

Slika 5. Agregacija SMS poruka

Inicijalizator vraća samo novu instancu konačne poruke koja je prazna, jer nije dodat nijedan dio. Svaki put kada dođe novi dio poruke, agregator će biti pozvan sa tim dijelom i trenutnim rezultatom agregacije kao argumentima. Ono što agregator treba da uradi je da na osnovu tih argumenata vrati novi objekat rezultata agregacije. Time smo završili agregaciju dobivši još jednu instancu KonačnaPoruka. Rezultat agregacije je *KTable* koja ima *String* kao ključ i KonačnaPoruka kao vrijednost. Sada ovaj *KTable* možemo pretvoriti u *stream*, filtrirati kako bismo dobili samo potpuno sklopljene zapise instance KompletnaPoruka i izvezli ih na izlaznu temu.

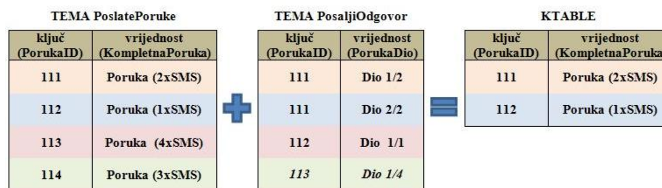
Poruke su ravnomerno kružno upisuju na particije izborom *null* ključa i raspodijeljene su po tematskim particijama, pa je time postignuta maksimalna moguća brzina obrade.

Budući da su tipovi ključeva izlazne teme i agregacijskog ključa različiti, potrebno ie transformirati ključeve zapisa prije korištenja operacije *to()*. Za postizanje ove transformacije koristimo *map()* operaciju i transformiramo naš ključ u *null* vrijednost tipa *Long*. Sada u izlaznoj temi imamo samo potpuno sklopljene instance kompletene poruke koje će biti obrađene drugom aplikacijom za upravljanje toka.

Nakon slanja SMS poruka provajderima servisa, potrebno je napraviti aplikativnu logiku koja će da prati status isporuke. Provajder telekomunikacionog servisa odnosno SMSC nam šalje takozvani "odgovor na dostavu" koji potvrđuje da je SMS poruka primljena i prihvaćena za daljnju obradu na strani operatera.

Da bismo podržali duge SMS poruke, imamo apstrakciju KompletnaPoruka, odnosno Klasa koja predstavlja dugu sms poruku zove se KompletnaPoruka. Takođe imamo i dijelove poruka, koji su stvarne SMPP poruke koje se šalju provajderu. Klasa koja predstavlja ove dijelove zove se PorukaDio.

KompletnaPoruka može predstavljati ili dugu poruku ili kratku poruku. Dijelovi poruke su SMPP poruke koje se stvarno šalju, a status cijele poruke zavisi od statusa svih dijelova poruke. To znači da za svaki odgovor na slanje koji stiže za određeni dio, moramo provjeriti da li su i svi drugi dijelovi poslali odgovore. Početni status je "PENDING", a ako su stigli odgovori za sve dijelove i statusi potvrđeni, možemo ažurirati status instance KompletnaPoruka na "SENT". Sve ovo se obavlja pomoću API u nekoliko koraka. Sl. 6. ispod ilustruje ono što je urađeno.


TEMA PoslatePoruke

ključ (PorukaID)	vrijednost (KompletnaPoruka)
111	Poruka (2xSMS)
112	Poruka (1xSMS)
113	Poruka (4xSMS)
114	Poruka (3xSMS)

TEMA PosaljiOdgovor

ključ (PorukaID)	vrijednost (PorukaDio)
111	Dio 1/2
111	Dio 2/2
112	Dio 1/1
113	*Dio 1/4*

KTABLE

ključ (PorukaID)	vrijednost (KompletnaPoruka)
111	Poruka (2xSMS)
112	Poruka (1xSMS)

Slika 6. Način i tok obrade VAS SMS poruka

Prilikom slanja instance KompletnaPoruka, postavljamo status na PENDING i pohranjujemo ih u Kafka Temu sa nazivom PoslatePoruke, s tim da je ključ KompletnaPorukaID generisan našom aplikacijom, a vrijednost je sama instanca KompletnaPoruka. Potom u zavisnosti od dužine poruke dijelimo instance KompletnaPoruka u jednu ili više instanci PorukaDio i šaljemo ih kao SMPP poruke odgovarajućem provajderu. Kada za specifičnu SMPP poruku dobijemo odgovor, on sadrži referencu na instancu PorukaDio koju smo poslali i ID koji je generisao provajder. Čuvanje instance PorukaDio u memoriji između slanja i primanja odgovora na predaju vrši SMPP biblioteka. U našem slučaju koristimo Cloudhopper biblioteku [9]. Kada dobijemo odgovor na dostavu, ažuriramo odgovarajuću PorukaDio instancu sa ID-om generisanog od strane provajdera i pohranimo je u Kafka temu pod nazivom PosaljiOdgovor sa ID-jem instance KompletnaPoruka kao ključem i vrijednošću PorukaDio.

Upotrebom KafkaStreams API-ja agregiramo instance PorukaDio koje pripadaju istom objektu KompletnaPoruka iz Teme PosaljiOdgovor i filtra za agregaciju svih dijelova. Rezultat ove operacije je *KTable* koja ima ključ ID PorukaDio instance i PorukaDioAgregirano kao vrijednost. Povezivanjem ovog *KTable* sa *KTable*-om kreiranim iz Teme PoslatePoruke dobijamo referencu na instance KompletnaPoruka u Temi PosaljiOdgovor i time znamo da su svi dijelovi poruke uspješno poslani i možemo ažurirati njen status na SENT. Da bi spojili dva *KTable* objekta, oni moraju imati isti ključ, pa smo koristili ID instance KompletnaPoruka kao ključ u Temama PoslatePoruke i PosaljiOdgovor. Ovim je pojašnjena logika procesiranja poruka, a u sistema je još implementiran Kafka rebalanser koji prikuplja informacije o trenutno aktivnim brokerima i provjerava da li su veze ravnomerno raspoređene, a detalji su dio posebne analize.

IV. POKRETANJE APLIKACIJE I TESTOVI PERFORMANSI

Za pokretanje Kafka klastera imali smo na raspolaganju intel Xeon 2530 v3 procesore, 3 VM sa 8 CPU, 16GB RAM, 15K SAS diskovima, 10G ethernet. Pokrenute su VM virtualizovane na KVM u OpenStack oblaku i sa Linux OS.

Pokrenuta je klaster sa tri brokera. Za potrebe testiranja performansi klastera pokrenute su Teme sa jednom i sa tri particije i sa isto toliko replika. Testiranje našeg klastera je urađeno prema [10] . Primjer kreiranja tema:

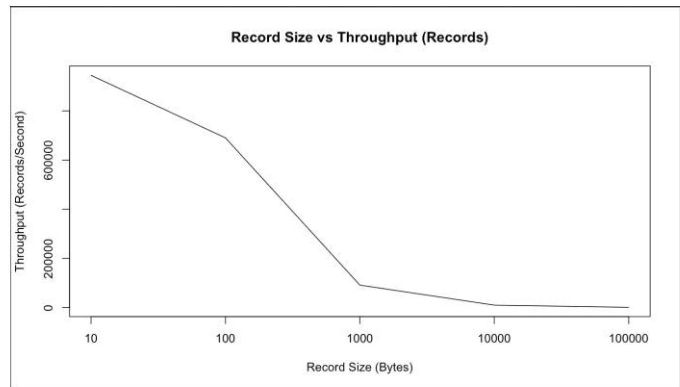
```
./kafka-topics.sh --zookeeper kafka-node1:2181 --create --topic MessageSEND --partitions 3 --replication-factor 1
```

```
./kafka-topics.sh --zookeeper kafka-node1:2181 --create --topic MessageNOTIFY --partitions 3 --replication-factor 3
```

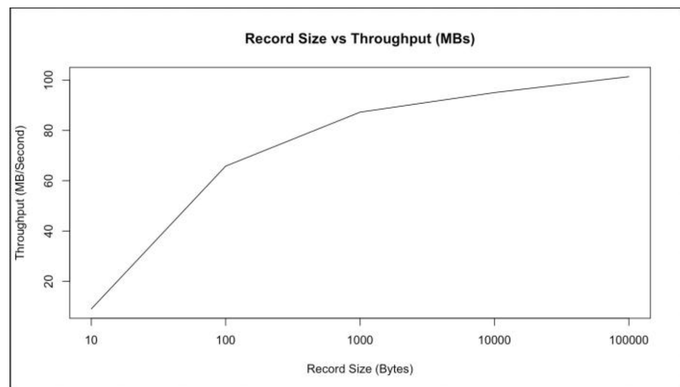
Navodimo rezultate nekoliko testova koji su interesantni kada je u pitanju VAS SMS saobraćaj i performanse predloženog klastera sa tri noda. Testovi pokazuju da preporučena konfiguracija pruža visoku propusnost i na strani Proizvođača i na strani Potrošača.

Uradili smo test propusnosti Proizvođača i Potrošača sa prethodno kreiranim temema. Faktor replikacije ne utiče na ishod ovog testa propusnosti Potrošača jer on čita samo iz jedne replike. Isto tako, ni nivo potvrde Proizvođača nije važan jer potrošač samo čita potpuno potvrđene poruke (čak i ako Proizvođač ne čeka punu potvrdu). Ovo osigurava da svaka poruka koju Potrošač vidi uvijek bude prisutna nakon pada kontrolora.

U testu ćemo konzumirati 50 miliona poruka u jednoj niti iz naše 3 particije i 3 puta replicirane teme. Kafkin Potrošač je veoma efikasan. Funkcioniše tako što preuzima dijelove loga direktno sa fajl sistema. On koristi *Sendfile API* [11] da ih prenese direktno kroz operativni sistem bez opterećenja kopiranja tih podataka kroz aplikaciju. Ovaj test zapravo počinje na početku loga, tako da radi realno I/O čitanje. Međutim, u produkcijskom okruženju, potrošač čita gotovo isključivo iz OS keša, pošto čita podatke koje je napisao Proizvođač i oni nalaze u kešu, Sl.7. i Sl. 8.



Slika 7. Arhitektura ZooKeeper klastera



Slika 8. Arhitektura ZooKeeper klastera

Test uključuje jednog Proizvođača koji koristi async replikaciju i tri Potrošača.

```
./kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance test 50000000 100 -1 acks=1 bootstrap.servers=kafka1-zoo:9092 buffer.memory=67108864 batch.size=8196
```

Na sva tri brokera potrebno je pokrenuti:

```
./kafka-consumer-perf-test.sh --zookeeper kafka1-zoo:2181 --messages 50000000 --topic test --threads 1
```

Ponovili smo test nekoliko puta i dobili smo propusnost cca 680K zapisa/s i obradu oko 65 MB/s na strani Potrošača što je gornja procesna granica.

Testovi su formulisani za prikaz performansi sa malim 100-bajtnim porukama, što nam je i bio cilj jer je standardni SMS veličine 128 bajta. Manje poruke su veći problem za sisteme za razmjenu poruka jer unose *overhead* što se može pokazati grafički. Propusnost je prikazana u zavisnosti od broja zapisa/sekundi i MB/sekundi tako što u petlji variramo veličinu zapisa/poruka. Grafikon pokazuje da se broj zapisa koje možemo poslati u sekundi smanjuje kako se veličina poruka povećava Sl. 7. Ali ako pogledamo propusnost u MB/sekundi, vidimo da se ukupni protok stvarnih korisničkih podataka povećava kako poruke postaju veće Sl. 8.

```
for i in 10 100 1000 10000 100000;do echo"" echo $i

./kafka-run-class.sh
org.apache.kafka.clients.tools.ProducerPerformance
test $((1000*1024*1024/$i)) $i -1 acks=1
bootstrap.servers=kafka1-zoo:9092
buffer.memory=67108864 batch.size=128000 done;
```

Možemo da vidimo da smo sa 10-bajtnim porukama opteretili CPU zaključavanjem i stalnim prebacivanjem poruka za slanje u redove i nismo u mogućnosti da maksimalno iskoristimo propusnost mreže. Počev od 100 bajtova, zapravo vidimo zasićenje mreže, ali i da propusnost još uvijek raste.

Izvršili smo i testiranje kašnjenja u isporuci poruka, odnosno test koliko dugo je potrebno da se poruka dostavi Potrošaču. U ovom testu više puta uzastopno mjerimo koliko vremena je potrebno Proizvođaču da pošalje poruku Kafka klasteru do trenutka kada je zaprimi Potrošač. Rezultati su se kretali u rasponu od 3ms do 20ms.

```
./kafka-run-class.sh kafka.tools.TestEndToEndLatency
kafka1-zk:9092 kafka2-zk:2181 test 5000
```

V. Zaključak

Izbor Kafke se pokazao kao odlično rješenje za telekomunikacione sisteme koji razmjenjuju veliku količinu poruka. Sistem radi veoma brzo i ima veliku propusnost i u konfiguracji sa minimalnim klasterom. Kašnjenja u obradi ulaznih VAS SMS podataka su zanemarljiva. Aplikativni dio se jednostavno implementira primjenom KafkaStrrems API i Kafka Java biblioteka. Izvršena je uspješna demonstracija upotrebe Apache Kafka u procesiranju SMS saobraćaja. U odnosu na referentno rješenje koje koristi Logica SMPP biblioteku i AMQ broker, brzina obrade i propusnost su višestruko veći. Testovi pokazuju da preporučena konfiguracija pruža visoku performanse i na strani Proizvođača i na strani Potrošača, uz zanemarljivo kašnjenje. Budući rad je zasnovan na primjeni predloženog rješenja i ELK steka u sistemu za obradu poruka. Integracija predloženog rješenja u postojeći sistem za obradu poruka jeste zahtjevna ali bi se na taj način u potpunosti prikazala njegova upotrebna vrijednost.

Literatura

[1] 3rd Generation Partnership Project specification, http://www.3gpp.org/ftp/tsg_t/tsg_t/tsgt_04/docs/pdfs/tp-99128.pdf,

[2] Gwen Shapira, Neha Narkhede, Todd Palino „Kafka: The Definitive Guide, Real-Time Data and Stream Processing at Scale - Second Edition", published by O'Reilly Media, September 2017.

[3] Nishant Garg „Apache Kafka Paperback", published by Packt Publishing Limited, October 2013.

[4] DataFlair Team, "https://data-flair.training/blogs/advantages-and-disadvantages-of-kafka/", published April 2018.

[5] Raul Estrada, "Apache Kafka Quick Start Guide: Leverage Apache Kafka 2.0 to simplify real-time data processing for distributed applications Paperback", published by Packt Publishing, December 2018.

[6] Manish Kumar, Chanchal Singh, "Building Data Streaming Applications with Apache Kafka: Design, develop and streamline applications using Apache Kafka", published by Packt Publishing Limited, August 2017.

[7] Apache Kafka Streems DSL Developer guide.," https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html/", 2017.

[8] Ellen Friedman, Ted Dunning, "Streaming Architecture: New Designs Using Apache Kafka and MapR Streams 1st Edition", published by O'Reilly Media, May 2016.

[9] Maven repository, Cloudhopper SMPP library, "https://mvnrepository.com/artifact/com.cloudhopper, 2015.

[10] Technical Blog, Software Testing Manual And Automation, "http://linuxthrill.blogspot.com/2016/04/kafka-benchmark-commands.html, October 2017.

[11] Sathish Palaniappan, Pramod Nagaraja, "Efficient data transfer through zero copy", published by IBM, September 2008.

Abstract

This paper represents implementation of messaging gateway broker that processing large number of VAS messages using Apache Kafka technology. It provides an optimized messaging solution that runs on OpenStack Cloud. It is based on Apache Kafka that combines stream processing and clustering in its algorithm and takes care of replication, redundancy and scaling of nodes within Kafka Cluster. The usege of Kafka Cluster increase the horizontal scalability and processing capacity of the messaging gateway system. The system is designed to serve for a fast, reliable and mutual exchange of VAS messages between external customers and telecommunication service providers. The obtained solution is significantly increased processing messaging capacity compared to use of AMQ broker. The proposed solution should to overcome the costs of system licensing because it is fully based on stable opensource components. The core of the system represents Apache Kafka cluster on a virtualized platform. It provides parallel processing of data and better performance compared to a referent messaging system. Using of Kafka cluster and appropriate configuration of the brokers guarantees the delivery of messages to the respective customers. We use ZooKeeper on server side and the Kafka Streams API client application-building library. Kafka Streams API has enabled us to combine the simplicity of coding using standard Java and Scala applications on the client side and advantages of Kafka's cluster technology on the server side.

Keywords- Kafka, Kafka Streems API, Zookeeper, Kafka DSL

USING APACHE KAFKA FOR MODELING THE MESSAGE FLOW PROCESSING SYSTEM

Nebojša Kuduz, Ozren Ćalić, Slaven Popović