Komunikacijski obrasci u mikroservisnoj arhitekturi

Dudaković, Timon

Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

TIMON DUDAKOVIĆ

KOMUNIKACIJSKI OBRASCI U MIKROSERVISNOJ ARHITEKTURI

Diplomski rad

Pula, rujan, 2022. godine

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

TIMON DUDAKOVIĆ

KOMUNIKACIJSKI OBRASCI U MIKROSERVISNOJ ARHITEKTURI

Diplomski rad

JMBAG: 0303076148, redoviti student
Studijski smjer: Informatika
Predmet: Izrada informatičkih projekata
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Nikola Tanković

Pula, rujan, 2022. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani_____, kandidat za magistra _____ ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljeni način, odnosno daje prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____

IZJAVA O KORIŠTENJU AUTORSKOG DJELA

Ja,_____dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom

_____

_____

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____

Potpis

_____

KOMUNIKACIJSKI OBRASCI U MIKROSERVISNOJ ARHITEKTURI

Timon Dudaković

Sažetak: Jedan od najvećih izazova pri prijelazu s aplikacije temeljene na monolitnoj arhitekturi na aplikaciju temeljenu na mikroservisnoj arhitekturi je usvajanje promjena prisutnih u komunikacijskoj paradigmi. Prijelaz iz poziva lokalnih metoda u nepouzdane sinkrone i asinkrone pozive između servisa dodaje višu razinu složenosti i smanjuje učinkovitost komunikacije što narušava performanse u raspodijeljenim sustavima. Izazovi dizajniranja i implementacije raspodijeljenog sustava su dobro poznati, ali je proces još uvijek dugotrajan i složen. Rješenje predstavljeno u ovom diplomskom radu uključuje visoke razine izolacije mikroservisa korištenjem asinkronih komunikacijskih obrazaca između internih mikroservisa. Iako postoji niz mikroservisa koji komuniciraju preko sinkronih komunikacijskih protokola, oni ne narušavaju integritet komunikacije i održavaju određenu razinu izolacije.

Ključne riječi: mikroservisi, raspodijeljeni sustavi, sinkroni i asinkroni komunikacijski obrasci, *service mesh*, *API Gateway*, obrasci otpornosti

COMMUNICATION PATTERNS IN A MICROSERVICE ARCHITECTURE

Timon Dudaković

Abstract: One of the biggest challenges when migrating from an application based on a monolothic arhictecture to an application based on a microservice architecture is adopting changes present in communication paradigm. Converting from local method calls to unreliable cross-service synchronous and asynchronous calls adds a higher level of complexity and reduces efficiency in communication which violates peformance in distributed systems. The challenges of designing and implementing a distributed system are well known, but the process is still long-lasting and complex. The solution presented in this thesis involves high levels of microservice isolation through the use of asynchronous communication patterns between the internal microservices. Although several microservices are communicating over synchronous communication protocols, they don't violate the integrity of the communication, and they maintain a certain level of isolation.

Keywords: microservices, distributed systems, synchronous and asynchronous communication patterns, service mesh, API gateway, resiliency patterns

Sadržaj

GitHub projekt: https://github.com/dudakovict/social-network

1. Uvod

Arhitekturni stil mikroservisa pristup je razvoju aplikacije kao skupu slabo povezanih manjih servisa (mikroservisa), od kojih se svaki izvodi u vlastitom okruženju i najčešće komunicira s vanjskim svijetom putem aplikacijskog sučelja (eng. *Application Programming Interface, API*) preko protokola *HTTP* (eng. *Hypertext Transfer Protocol*) [1]. Mikroservis je u ovom kontekstu autonomna, nezavisna aplikacija koja komunicira putem mreže. Mikroservisi ne moraju poznavati implementacijske detalje mikroservisa s koji komuniciraju, već je dovoljno poznavati strukturu aplikacijskog sučelja servisa s kojim se ostvaruje komunikacija. Snažan razlog za usvajanje mikroservisne arhitekture je smanjenje ovisnosti između mikroservisa što omogućuje jednostavnije upravljanje greškama, bržu distribuciju novih verzija mikroservisa i njihovo skaliranje [2]. Omogućeno je ažuriranje mikroservisa bez ponovne distribucije cijele aplikacije, te vraćanje mikroservisa na prethodno stanje ako dođe do nepredviđenih grešaka. Također, mikroservisi se mogu skalirati nezavisno, što omogućuje skaliranje podsustava koji zahtijevaju više resursa, bez skaliranja cjelokupne aplikacije [2].

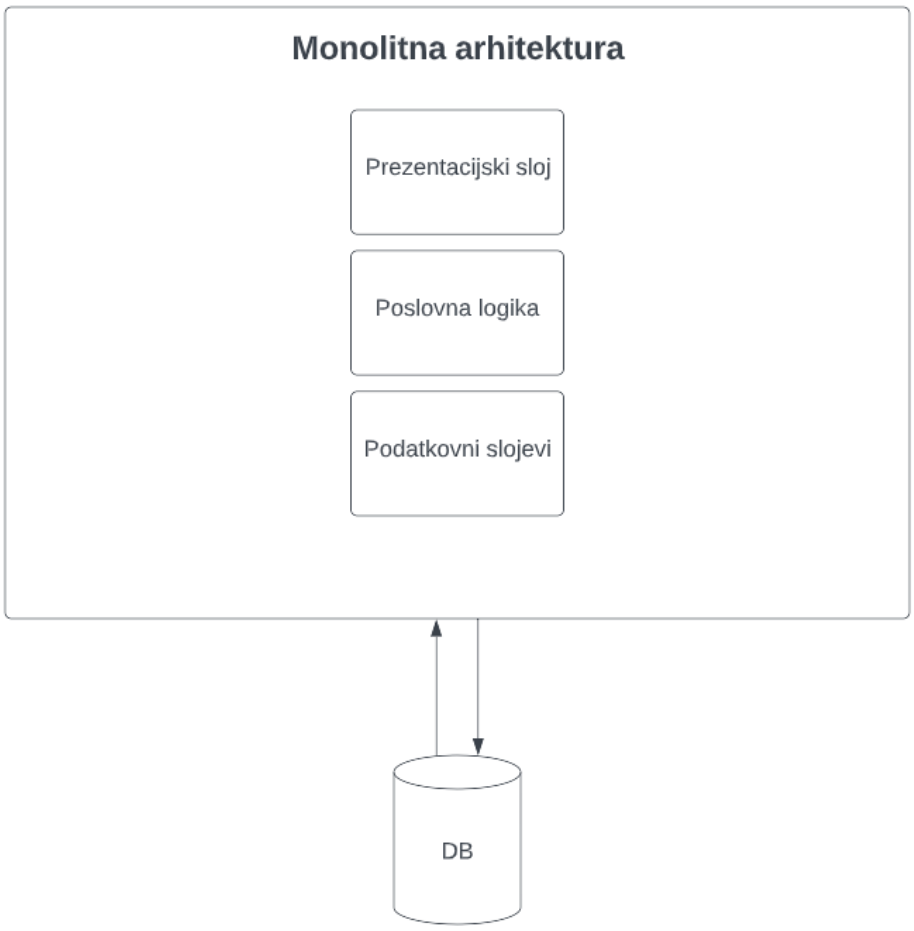
Kao i bilo koji drugi arhitekturni stil, mikroservisi nisu savršeni, imaju svoje prednosti i nedostatke. Neke od prednosti su jednostavnije skaliranje pojedinih aspekata sustava ovisno o opterećenju, i mogućnost suradnje veće količine programera ili timova na istom sustavu zbog određenih karakteristika mikroservisne arhitekture, kao što su izolacija podataka i grešaka te mogućnost korištenja različitih tehnologija [1]. Međutim, kako se povećava količina mikroservisa, tako se povećava i složenost njihovog održavanja, distribucije novih verzija i nadziranja. Još jedan aspekt kod kojeg se povećava složenost je komunikacija. Postoji više načina komunikacije između mikroservisa, i njen odabir ovisi o zahtjevima nametnutim na sustav, kao što su dozvoljena kašnjenja i tolerancije na greške. Odabir komunikacijskog mehanizma ovisi o tome da li je komunikacija između mikroservisa jednosmjerna ili dvosmjerna, da li sustav zahtijeva visoku razinu nezavisnosti mikroservisa, ili da li odgovor na pojedini zahtjev mora obraditi isti mikroservis koji je inicirao komunikaciju [3].

2. Monolitna i mikroservisna arhitektura

Kako bi započeli s detaljnijim opisom mikroservisne arhitekture, korisno je usporediti ga s monolitnom arhitekturom. Ovo poglavlje predstavlja uvod u monolitne i mikroservisne arhitekturne paradigme u kojoj će se izložiti njihove karakteristike, prednosti i nedostatci.

2.1. Monolitna arhitektura

Monolitna arhitektura se odnosi na stil razvoja aplikacija koja je izgrađena kao zapakirana jedinica za koju kažemo da je samostalna i nezavisna [1]. Riječ "monolit" često se pripisuje nečem velikom, što nije daleko od istine o monolitnoj arhitekturi kao stilu za dizajn aplikacije. Najčešće je sastavljena tako da su prezentacijski sloj i poslovna logika sadržani unutar jednog repozitorija koda i jedne komponente što se vidi na slici *Slika **1***.



Slika 1: Monolitna arhitektura

To ne znači da unutar svakog sloja ne može postojati nekoliko modula od kojih svaka služi određenoj funkciji, nego samo da su svi slojevi povezani unutar jedne aplikacije. S obzirom na to da su sve funkcionalnosti povezane i ukomponirane na istoj instanci, donošenje velikih promjena na sustavnoj razini je jednostavnije nego što bi bilo da je sva funkcionalnost razgranata kroz više zasebnih servisa [1]. Budući da se interna komunikacija može izvršavati jednostavnim funkcijskim pozivima, nije potrebno brinuti se o internoj kompatibilnosti aplikacijskog sučelja. Prednost funkcijskih poziva kao načina komunikacije je što osigurava minimalne troškove i kašnjenja. Orkestracija distribucije aplikacije je jednostavna jer najčešće postoji samo jedna centralna ulazna točka u program. Budući da se komponente i njihove promjene distribuiraju u isto vrijeme, bilo kakav neuspjeh tijekom distribucije rješava se jednostavnim vraćanjem verzije aplikacije na prethodno stanje. Također, ne postoje problemi s ovisnostima unutar aplikacije s obzirom na to da su troškovi interne komunikacije minimalni i ovisnosti uvijek dostupne. To znači da aplikacija ne može stvarno biti neispravna bez i da je komponenta koja inicira komunikaciju neispravna, jer su dio iste instance aplikacije [1].

Primarna prednost monolitne arhitekture je velika brzina razvoja zato što se sve komponente sustava nalaze na jednom mjestu. Uz to pojednostavljujemo i samu distribuciju aplikacije jer postoji samo jedna ulazna točka u program. Također, kod centralizirane aplikacije, jedno aplikacijsko sučelje često može obavljati istu funkciju koju bi kod mikroservisne arhitekture zahtijevala prolaženje kroz više mikroservisa. Samim time je proces testiranja i otklanjanja grešaka puno brži nego kod raspodijeljene aplikacije.

Prednost	***Opis***
Manje međusektorskih briga	Međusektorski problemi su problemi koji utječu na cijelu aplikaciju kao što su upravljanje bilježenjem, predmemoriranjem i nadziranjem [2]. Budući da se svi navedeni međusektorski problemi nalaze na jednom mjestu omogućuje jednostavnije upravljanje s njima.

Jednostavnije otklanjanje grešaka i testiranje	Za razliku od mikroservisne arhitekture, proces otklanjanja grešaka i testiranja kod monolitne aplikacije je puno jednostavniji. Budući da je monolitna aplikacija jedna nedjeljiva jedinica, možemo pokretati testove na razini sustava puno brže [2].
Jednostavna distribucija	Kada je riječ o monolitnim aplikacijama, ne moramo se baviti distribucijom na razini servisa jer se sve komponente i funkcionalnosti nalaze unutar jedne datoteke ili direktorija [2].
Jednostavan razvoj	Monolitne aplikacije imaju nižu razinu složenosti, pa se brže razvijaju. Osim toga, moguće je započeti s osnovnom aplikacijom i zatim dodavati značajke kako se razvija.

Tablica 1: Prednosti monolitne arhitekture

Međutim, monolitna arhitektura ima i svoje nedostatke. Jednostavna i jeftina komunikacija između internih komponenti može dovesti do visoko povezanog i isprepletenog grafa ovisnosti [1]. Iako njegova degradacija nema toliko visoku cijenu u monolitnoj arhitekturi, i dalje može utjecati na brzinu razvoja jer bilo kakve promjene imaju veće šanse propagirati daljnje potrebe za promjenama u sustavu. Stoga, greška u jednoj komponenti može narušiti integritet cijelog sustava. To može značiti da greška u relativno nevažnoj komponenti može prekinuti ili pogoršati razinu integriteta cijelog sustava. Monolitna aplikacija također može uzrokovati probleme tijekom njenog skaliranja. Timovi koji rade na istoj aplikaciji si međusobno mogu izazvati dodatan posao radi promjena koje donose u sustav. Unatoč navedenom, monolitne aplikacije mogu biti izvrsno rješenje za mnoga okruženja i slučajeve uporabe. Budući da sadrži sve komponente potrebno za njenu inicijalizaciju, dobro strukturiranu monolitnu aplikaciju jednostavno je razumjeti, razvijati i distribuirati. Također, činjenica je da većinu svoje komunikacije može provoditi interno, što rezultira time da možemo izbjeći mnoge izazove mrežne komunikacije, poput latencija i kvarova na mreži [1].

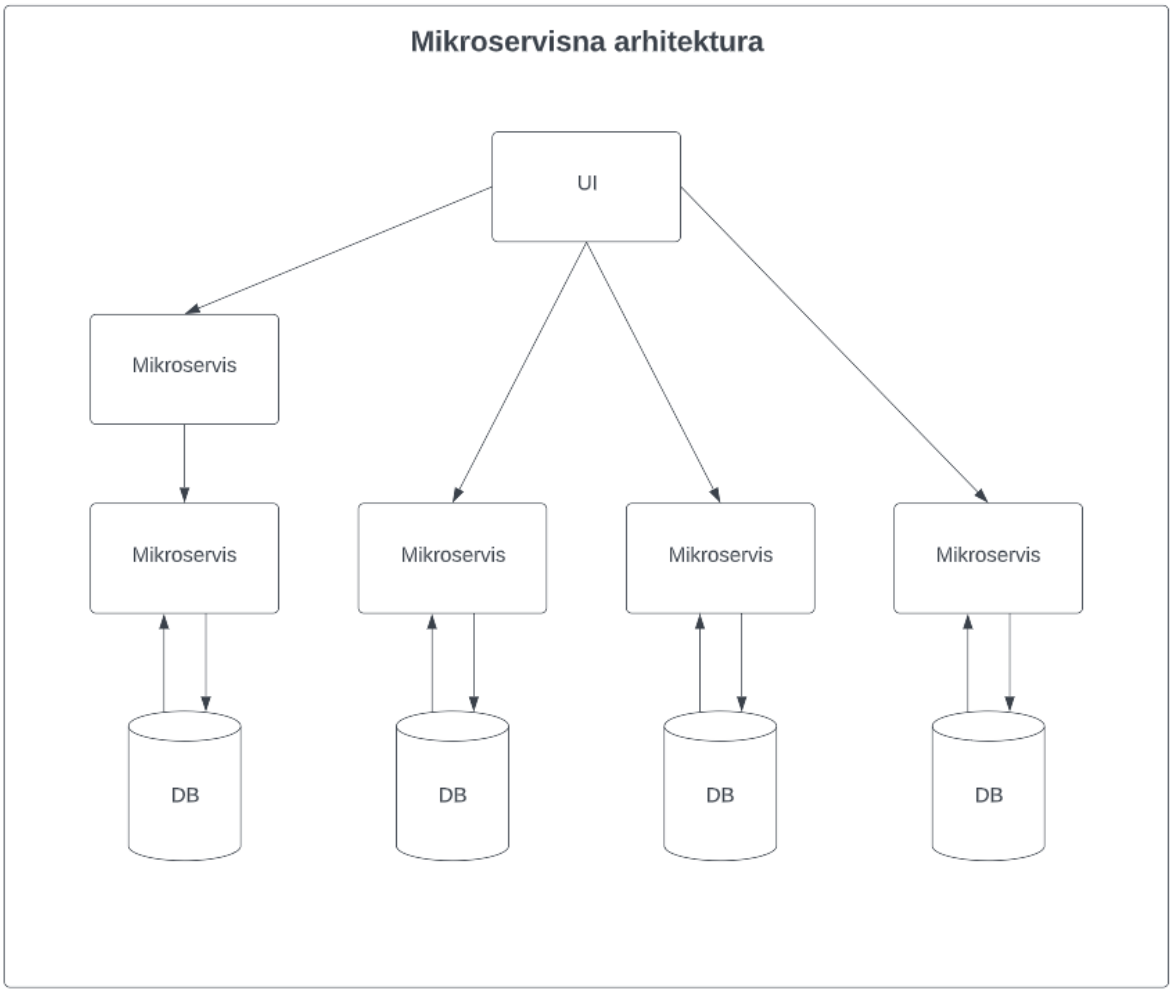
Nedostatak	***Opis***
Složenost	Kada se monolitna aplikacija skalira, postaje prekomplicirana za razumijevanje. Također, teško je upravljati složenim sustavom unutar jedne aplikacije.
Donošenje promjena	Teže je implementirati promjene u velikoj i složenoj aplikaciji. Svaka promjena utječe na cijeli sustav pa se isti mora temeljito uskladiti. To čini cjelokupni razvojni proces dugotrajnim.
Skalabilnost	Ne možemo skalirati komponente zasebno, samo cijelu aplikaciju.
Tehnološke barijere	Iznimno je problematično primijeniti novu tehnologiju u monolitnoj aplikaciji jer se tada cijela aplikacija mora ponovno pisati.

Tablica 2: Nedostatci monolitne arhitekture

Monolitne aplikacije mogu biti prilično učinkovite sve dok ne postanu prevelike za održavati i skaliranje ne postane izazov. Uvođenje malih promjena u jednoj komponenti zahtijeva ponovnu izgradnju i testiranje cijelog sustava. Samim time, što aplikacija postaje veća i složenija, brzina razvoja se smanjuje. Uz navedene nedostatke, javljaju se i problemi koji utječu na pouzdanost i fleksibilnost same aplikacije. Ako postoji greška u bilo kojem dijelu aplikacije, to može narušiti dostupnost cijelog sustava. Shodno tome, bilo kakve promjene u korištenim alatima i tehnologijama utječu na cijelu aplikaciju, što znači da bilo kakvo uvođenje promjena postaje skupo i dugotrajno [1].

2.2. Mikroservisna arhitektura

Mikroservisna arhitektura, također poznata kao mikroservisi, su arhitektonski i organizacijski pristup razvoju aplikacije gdje se aplikacija sastoji od malih, slabo povezanih, autonomnih servisa koji komuniciraju preko dobro definiranih aplikacijskih sučelja [1]. Cilj mikroservisa je razviti nizak stupanj povezanosti i visok stupanj kohezije između servisa. Kada govorimo o razini povezanosti među servisima, mislimo na količinu ovisnosti koje postoje između njih, dok kohezija predstavlja razinu povezanosti komponenti unutar servisa [2]. Visoka povezanost može naznačiti da je potrebno ponovno promisliti kontekst i domenu usluge servisa, a niska kohezija da se funkcionalnost treba podijeliti na zasebne, smislene servise jer nemaju određenu razinu korelacije da bi se nalazili unutar istog [2]. Na slici *Slika **2*** prikazan je izgled visoke razine mikroservisne arhitekture.



Slika 2: Mikroservisna arhitektura

Mikroservisi donose mnoge organizacijske prednosti kao što su jednostavnije upravljanje greškama, učinkovitije skaliranje i veća brzina kod razvoja. Također, omogućuje korištenje heterogenog tehnološkog skupa kako bi programeri imali veću slobodu odabira ovisno o potrebama, za razliku od monolitne arhitekture, koja obično nameće homogeni tehnološki skup [4]. Nadalje, mikroservisi omogućuju nezavisno skaliranje servisa. Različiti servisi mogu imati različite potrebe za resursima, pri čemu je nekima potrebno više memorije, a nekima više procesne snage. Budući da servisi mogu zahtijevati različite vrste resursa, možemo ih učinkovitije rasporediti prema potrebi što dovodi do manjih infrastrukturnih troškova. Uz navedeno, mikroservisi pojednostavljuju i skaliranje razvojne organizacije. Za svaki servis je odgovoran određeni tim, a oni se integriraju samo putem specificiranih i dogovorenih aplikacijskih sučelja. Samim time, jasnije je tko je vlasnik i tko posjeduje vlasništvo nad podacima svakog servisa unutar cijelog sustava. Budući da mikroservisi sačinjavaju funkcionalnost cijelog sustava od skupa malih servisa, iste će biti puno jednostavnije distribuirati nego što bi bilo kod ekvivalentne monolitne aplikacije, koja bi imala puno više pokretnih dijelova. Shodno tome, korištenjem odgovarajućih sustava za nadziranje i bilježenje možemo biti puno precizniji i sigurniji tijekom detekcije grešaka i kvarova u servisima.

Prednost	***Opis***
Nezavisne komponente	Moguće je distribuirati i ažurirati sve servise nezavisno, što nam pruža veću fleksibilnost. Uz to, greška u jednom servisu ne utječe na cijelu aplikaciju, već je izolirana samo na taj servis. Također, puno je jednostavnije dodati nove značajke u mikroservisnoj aplikaciji nego monolitnoj.
Jednostavnije razumijevanje	Budući da je mikroservisna aplikacija podijeljena na manje i jednostavnije komponente, jednostavnija je za razumjeti i upravljati njome. Možemo se koncentrirati samo na određen servis koji

	je povezan s poslovnim problemom koji pokušavamo riješiti.
Poboljšana skalabilnost	Moguće je skalirati svaki element nezavisno, što je puno učinkovitije od skaliranja monolitnih aplikacija.
Heterogeni tehnološki skup	Timovi nisu ograničeni tehnologijom odabranom od samog početka. Imaju slobodu primijeniti različite tehnologije i programske okvire za svaki servis.
Viša razina agilnosti	Svaka greška ili kvar utječe samo na određeni servis, a ne na cijeli sustav. Posljedično, sve promjene i eksperimenti provode se s manje grešaka i rizika.

Tablica 3: Prednosti mikroservisne arhitekture

Mikroservisi također donose mnoge izazove. Skup visoko ovisnih mikroservisa može izgledati kao raspodijeljeni monolit. To znači da svaka degradacija performansi u jednom servisu može rezultirati degradacijom performansi cijelog sustava, i bilo kakve greške u jednom servisu mogu rezultirati degradacijom ili potpunim zastojem funkcionalnosti u cijelom sustavu [1]. Nadalje, svaki zastoj servisa može napraviti puno jači negativan efekt u slučaju kada je razina povezanosti i ovisnosti između servisa jako visoka. U suprotnom, svaka degradacija performansi ili funkcionalnosti je izolirana samo na podskup servisa. S obzirom na to da su aplikacijska sučelja integracijske točke servisa, njihovo održavanje kompatibilnosti je ključno. Bilo kakve promjene u aplikacijskom sučelju moraju biti orkestrirane i dogovorene s timovima koji ih koriste sa svrhom održavanja kompatibilnosti [2].

Mikroservisna arhitektura uvodi cijeli drugi podskup problema kao što su upravljanje greškama, konkurencijom, međusektorskim problemima i komunikacijom. Kako bi se stvorila dosljedna mikroservisna arhitektura, važno je kod dizajna uzeti u obzir komunikacijske obrasce jer oni predstavljaju integracijske točke između servisa, i samim time određuju mnoge karakteristike cjelokupnog sustava, kao što su skalabilnost i tolerancije na greške [2]. Svaki dodatan servis s kojim je potrebno uspostaviti

komunikaciju je dodatna ovisnost servisa koji inicira komunikaciju, iz čega proizlazi da se automatski povećava i razina ovisnosti unutar sustava. Stoga je važno razmisliti s kime određeni servis treba komunicirati, kako, i što. Način komunikacije ima veliku ulogu kod definiranja razine ovisnosti koja će postojati između servisa.

Nedostatak	***Opis***
Dodatna složenost	Budući da je mikroservisna arhitektura raspodijeljeni sustav, moramo odabrati način komunikacije i uspostaviti vezu između svih servisa i baza podataka. Također, sve dok takva aplikacija sadrži nezavisne servise, svi oni moraju biti nezavisno distribuirani.
Distribucija sustava	Mikroservisna arhitektura je složen sustav sastavljen od više servisa i baza podataka. Stoga, moramo pažljivo pristupiti svakoj uspostavi veze i komunikacije.
Međusektorski problemi	Neki od bitnijih međusektorskih problema uključuju postavljanje vanjske konfiguracije, sustava za bilježenje, metrika, i provjera zdravlja sustava [4].
Testiranje	Mnoštvo servisa koji se mogu nezavisno distribuirati čini testiranje puno zahtjevnijim.

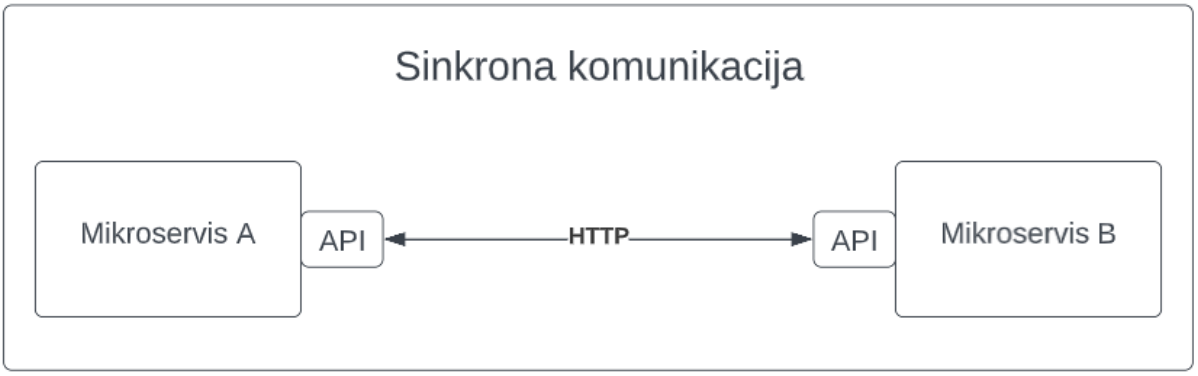
Tablica 4: Nedostatci mikroservisne arhitekture

3. Sinkrona i asinkrona komunikacija

Postoje dva glavna načina za uspostavu komunikacije u mikroservisnoj aplikaciji. Sinkrona, u kojoj se komunikacija odvija u stvarnom vremenu, i asinkrona, u kojoj se komunikacija odvija nezavisno o vremenu [3]. U ovom poglavlju ćemo predstaviti sinkrone i asinkrone komunikacijske obrasce i objasniti kada jedan način komunikacije ima prednost nad drugim.

3.1. Sinkrona komunikacija

Sinkrona komunikacija je način komunikacije koja se odvija u stvarnom vremenu, u kojoj svi sudionici sudjeluju istovremeno [5]. Strana koja inicira komunikaciju čeka suprotnu stranu da primi zahtjev, obradi ga, i vrati odgovor. Sinkrona komunikacija je najčešći način komunikacije, budući da je izravna, i jednostavna za razumjeti. Često se koristi pri dizajniranju sustava na višoj razini.



Slika 3: Sinkrona komunikacija između servisa

Prednost sinkrone komunikacije je njezina jednostavnost koncepta i implementacije. Možemo pratiti stanje zahtjeva prije i poslije odgovora, jer se ono nalazi u istom funkcijskom kontekstu [3]. Samim time, odgovor možemo logički obraditi na istom mjestu gdje je zahtjev iniciran. To nam omogućuje jednostavno praćenje tijeka komunikacije kroz program. Prema definiciji sinkrone komunikacije, također možemo izvršiti sinkrone pozive koristeći asinkrone obrasce ako čekamo na odgovor u istom procesu [5]. Takva vrsta poziva se naziva blokirajući poziv, jer blokira izvršenje preostalog dijela koda sve dok ne primimo odgovor. Možemo uzeti za primjer korištenje

sustava reda čekanja (eng. *queue*) za komunikaciju s nekakvim vanjskim servisom. On nam omogućuje uporabu asinkrone komunikacije, ali može postići i sinkronu u slučaju kada se proces koji je inicirao komunikaciju zaustavi, i blokira dok čeka na odgovor [5]. Stoga, unutarnji proces koji je inicirao komunikaciju i proces vanjskog servisa su aktivni u isto vrijeme.

Sinkrona komunikacija ne zahtijeva nužno dodavanje bilo kakvog potpornog servisa u sustav. Interni servisi mogu međusobno komunicirati putem jednostavnih mrežnih poziva. Iznimka je kada sustav zahtijeva jako visoku propusnost. U tom slučaju možemo uključiti dodatnu infrastrukturnu komponentu između servisa. Najčešće se koristi *load balancer* [5].

3.2. Sinkroni komunikacijski obrasci

Najpoznatiji sinkroni komunikacijski obrasci su *REST* (eng. *REpresentational State Transfer*) i *gRPC* (eng. *Google Remote Procedure Call*). Dok *REST* sučelje danas može koristiti *HTTP/2* ili *HTTP/1.1*, *gRPC* isključivo koristi *HTTP/2* kao temeljni transportni protokol [6]. Iako je *HTTP/1.1* dugo bio glavni način komunikacije na internetu, imao je brojne nedostatke koje je bilo potrebno zaobići. Neki od nedostataka su ograničeni broj veza na *host* mašini i osjetljivost na kašnjenja. Kao rješenje za navedene nedostatke, stvoren je *HTTP/2*. Jedna od najvećih promjena je dodana mogućnost za uspostavu višestrukih istovremenih tokova unutar iste veze, što je omogućilo jednoj vezi da istovremeno preuzima što više resursa sa poslužitelja. Rezultat toga je poboljšano ponašanje privremene memorije, i dodana mogućnost poslužitelja da preventivno šalje resurse prema klijentu u slučaju kada su dva određena resursa vrlo često tražena u istom zahtjevu [7]. Razlike na visokoj razine između *HTTP/2* i *HTTP/1.** su sljedeće:

- *HTTP/2* je binaran, umjesto tekstualan
- *HTTP/2* je potpuno multipleksiran, umjesto uređen i blokirajući
- *HTTP/2* može koristiti jednu vezu kako bi postigao paralelizam
- *HTTP/2* koristi kompresiju zaglavlja za smanjenje opterećenja
- *HTTP/2* omogućuje poslužiteljima proaktivno slanje odgovora u privremenu memoriju klijenta

3.2.1. Arhitekturni stil *REST*

REST je komunikacijski obrazac često korišten u kombinaciji s protokolom *HTTP*. Izvorni opis *REST* komunikacijskog obrasca definirao je četiri ograničenja koja aplikacijska sučelja moraju zadovoljiti, kako bi za određeno aplikacijsko sučelje mogli reći da je *RESTful*. Ta ograničenja uključuju identifikaciju resursa, manipulaciju resursa, samo opisne poruke i hipermedij kao pokretač aplikacijskog stanja [8]. *REST* koristi *HTTP* metode za definiranje operacija na resursima kojima određeni servis upravlja, i *HTTP* status kodove za označavanje rezultata operacije.

Status kod	***Opis***
1XX Informational	Komunicira informacije o prijenosu na razini protokola.
2XX Success	Indicira da je zahtjev klijenta uspješno primljen, shvaćen, i prihvaćen.
200 OK	Indicira da je zahtjev uspješno obrađen.
201 Created	Indicira da je zahtjev uspješno obrađen i kao rezultat je stvoren novi resurs.
204 No Content	Indicira da je zahtjev uspješno obrađen, ali ne vraća nikakav odgovor.
3XX Redirection	Indicira da klijent mora poduzeti dodatne radnje kako bi dovršio svoj zahtjev.
4XX Client Error	Indicira da je greška prouzročena sa strane klijenta.
400 Bad Request	Poslužitelj nije mogao razumjeti zahtjev zbog netočne sintakse.
401 Unauthorized	Indicira da zahtjev klijenta ne sadrži podatke o autentičnosti korisnika.
403 Forbidden	Neovlašteni zahtjev. Klijent nema prava pristupa resursu.
404 Not Found	Indicira da traženi resurs nije moguće pronaći.
5XX Server Error	Indicira da je greška prouzročena sa strane servera.
500 Internal Server Error	Server je naišao na neočekivani uvjet koji ga je spriječio da ispuni zahtjev.

Tablica 5: Klase HTTP status kodova i najčešće korišteni status kodovi

REST se temelji na resursima. Primjer resursa bi mogao biti proizvod, koji predstavlja sve proizvode poduzeća. Oni se mogu stvarati, čitati, ažurirati i brisati putem *REST* metoda. Tablica *Tablica **6*** prikazuje kako bi mogla izgledati definicija *REST* aplikacijskog sučelja za resurs proizvoda.

HTTP metoda	***Put upita***	***Opis***
GET	/proizvod	Vraća podatke o svim proizvodima.
GET	/proizvod/1	Vraća podatke o proizvodu čiji je *ID* 1.
POST	/proizvod	Stvara proizvod sa podacima iz tijela *HTTP* zahtjeva.
PUT	/proizvod/1	Stvara novi proizvod čiji je *ID* 1, ili zamjenjuje proizvod čiji je *ID* 1 sa podacima iz tijela *HTTP* zahtjeva.
PATCH	/proizvod/1	Ažurira podatke proizvoda čiji je *ID* 1 sa podacima iz tijela *HTTP* zahtjeva.
DELETE	/proizvod/1	Briše proizvod čiji je *ID* 1.

Tablica 6: Primjer definicije RESTful aplikacijskog sučelja za resurs proizvoda

3.2.2. Programski okvir *gRPC*

gRPC je moderan *RPC* (eng. *Remote Procedure Call*) programski okvir koji radi na protokolu *HTTP/2*. Kao svoj format za serijalizaciju strukturiranih podataka koristi *protocol buffer* [6]. Osim što omogućuje definiranje strukture podataka, također omogućuje definiranje servisa pomoću definiranih struktura podataka. U usporedbi s *REST* arhitekturnim stilom koji je dosta orijentiran na resurse, *gRPC* je više orijentiran na procedure i omogućuje definiranje aplikacijskog sučelja bez toliko ograničenja.

Protobuf (skraćeno za *protocol buffers*) je besplatan multi-platformski binaran format podataka koji se koristi za optimiziranu serijalizaciju strukturiranih podataka putem formata binarne žice (eng. *binary wire format*) [7]. Veličina poruke se može smanjiti do deset puta u usporedbi s formatom *JSON* (eng. *JavaScript Object Notation*).

Budući da su strukture podataka i servisi definirani pomoću jezika za opis sučelja (eng. *Interface Definition Language, IDL*), *gRPC* dopušta generiranje koda za klijente i poslužitelje. Za stranu klijenta, generiranje koda stvara potrebne vrste ulaza i izlaza, te definicije za metode koje pozivaju *gRPC* poslužitelj. Za stranu poslužitelja, stvaraju se potrebni tipovi podataka i sučelja za metode servisa koje se zatim mogu implementirati za stvaranje funkcionalnosti. Generiranje koda omogućuje brzo i jednostavno implementiranje poslužitelja, ili za stvaranje novog klijenta za interakciju sa poslužiteljem. Korištenje jezika za opis sučelja u kombinaciji s pristupom generiranja koda ima dodatnu prednost jer definira tipove podataka centralno u specifikacijskoj datoteci za korištenje od strane klijenta i poslužitelja što je korisno jer obje strane znaju tipove podataka koji se koriste u komunikaciji, što povećava samu sigurnost tipova podataka [7]. Na slici *Slika **4*** prikazan je mogući izgled definicije *gRPC* servisa za resurs proizvoda.

```
syntax = "proto3";

package product;

service ProductService {
  rpc CreateProduct(CreateProductRequest) returns (CreateProductResponse) {}
}

message CreateProductRequest {
  string name = 1;
  int64 price = 2;
}

message CreateProductResponse {
  int64 status = 1;
  string message = 2;
  int64 id = 3;
}
```

Slika 4: Definicija gRPC servisa za proizvod

Za oboje ulaz i izlaz, *gRPC* omogućuje definiranje jedne stavke, ili toka (eng. *stream*) stavki. Kao rezultat toga, *gRPC* omogućuje četiri različita komunikacijska obrasca koji se razlikuju po vrsti ulaza i izlaza koji mogu biti unarni ili tokovi [7]. U dvosmjernim tokovima, čitanje i pisanje se izvršava nezavisno, što znači da klijent i poslužitelj mogu čitati i pisati kako god žele. Jedan nedostatak *gRPC* komunikacijskog obrasca je što nije toliko dobro podržan u svim preglednicima kao što je na drugim platformama.

Tip	***Ulaz***	***Izlaz***
Unary RPC	Jedan zahtjev	Jedan odgovor
Server streaming RPC	Jedan zahtjev	Tok odgovora
Client Streaming RPC	Tok zahtjeva	Jedan odgovor
Bidirectional streaming RPC	Tok zahtjeva	Tok odgovora

Tablica 7: Vrste gRPC komunikacijskih obrasca

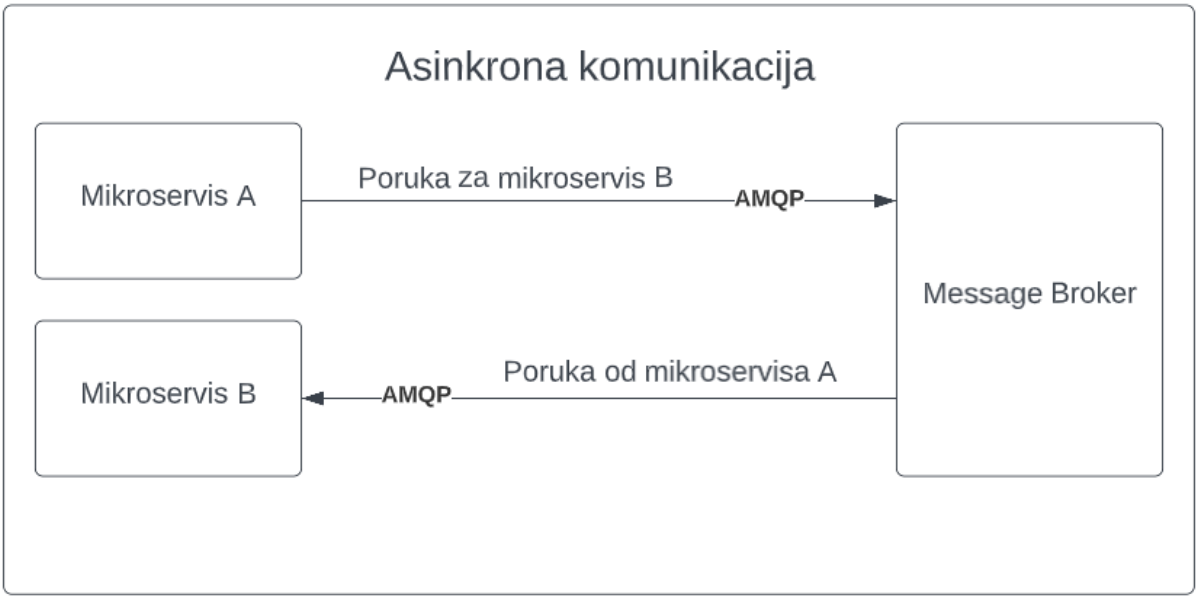
3.3. Asinkrona komunikacija

Asinkrona komunikacija je suprotna od sinkrone komunikacije jer se ne odvija istovremeno [3]. Dok su u sinkronoj komunikaciji pošiljatelj i primatelj uključeni u isto vrijeme, to ograničenje ne vrijedi za asinkronu komunikaciju. To znači da možemo privremeno razdvojiti pošiljatelja i primatelja, tako da primatelj nije obvezan obraditi podatke odmah kada stignu. Osim toga, odvajamo izravnu komunikaciju između njih, jer asinkrona komunikacija uobičajeno uključuje neku vrstu posrednika ili mehanizma reda čekanja između njih, što se vidi na slici *Slika **5***. Najčešće koristimo nekakav vanjski servis za obavljanje takvog posla.

Za asinkronu komunikaciju kažemo da je temeljena na "guranju" (eng. *push-based*) ili "povlačenju" (eng. *pull-based*) [2]. Kod asinkrone komunikacije temeljene na "guranju", posrednik "gura" poruke prema potrošaču kako bi se procesirale. To znači da je odgovornost posrednika osigurati da se poruke šalju potrošačima. U slučaju s više potrošača, odgovornost je posrednika pobrinuti se da poruke budu dostavljene svim potrebnim potrošačima. Kod asinkrone komunikacije temeljene na "povlačenju", odgovornost je potrošača inicirati komunikaciju s posrednikom da zatraži poruke za

procesiranje. To potrošaču daje potpunu kontrolu nad propusnosti, omogućujući mu da diktira tempo. To znači da potrošač općenito neće biti preopterećen poslom, osim ako procesiranje poruka ne uzrokuje heterogenu količinu posla. To bi moglo uzrokovati da pogrešno konfigurirani potrošač premaši svoje resurse ako primi veliku količinu zahtjevnih poruka koje pokušava obraditi paralelno.

Jedan važan pojam u asinkronoj komunikaciji je povratni pritisak (eng. *backpressure*) ili kontrola toka (eng. *flow-control*) [2]. Odnosi se na scenarij kada potrošač nije u stanju nositi se s propusnošću koje šalje proizvođač, te prisiljava proizvođača da uspori. Kod asinkrone komunikacije temeljene na "povlačenju", problem s propusnošću riješen je činjenicom da je potrošač taj koji kontrolira propusnost, ali kod asinkrone komunikacije temeljene na "guranju" ovo je vrlo korisna značajka jer omogućuje potrošaču da naznači da je preopterećen propusnošću, i uspori proizvođača. Druge alternative uključuju rješenja kao što su predmemoriranje (eng. *buffering*) ili odbacivanje poruka.



Slika 5: Asinkrona komunikacija između servisa

Asinkrona komunikacija može omogućiti otpornije i elastičnije arhitekture. Proizvođača i potrošača razdvaja posrednik, koji može čuvati poruke u međuspremniku dok se ne isporuče ili obrade. Proizvođač i potrošač mogu biti odvojeni na razini veze ili na razini vremena. Odvajanje na razini veze znači da potrošač i proizvođač nisu izravno povezani, već su razdvojeni posrednikom u sredini. To znači da potrošač ne mora biti dostupan

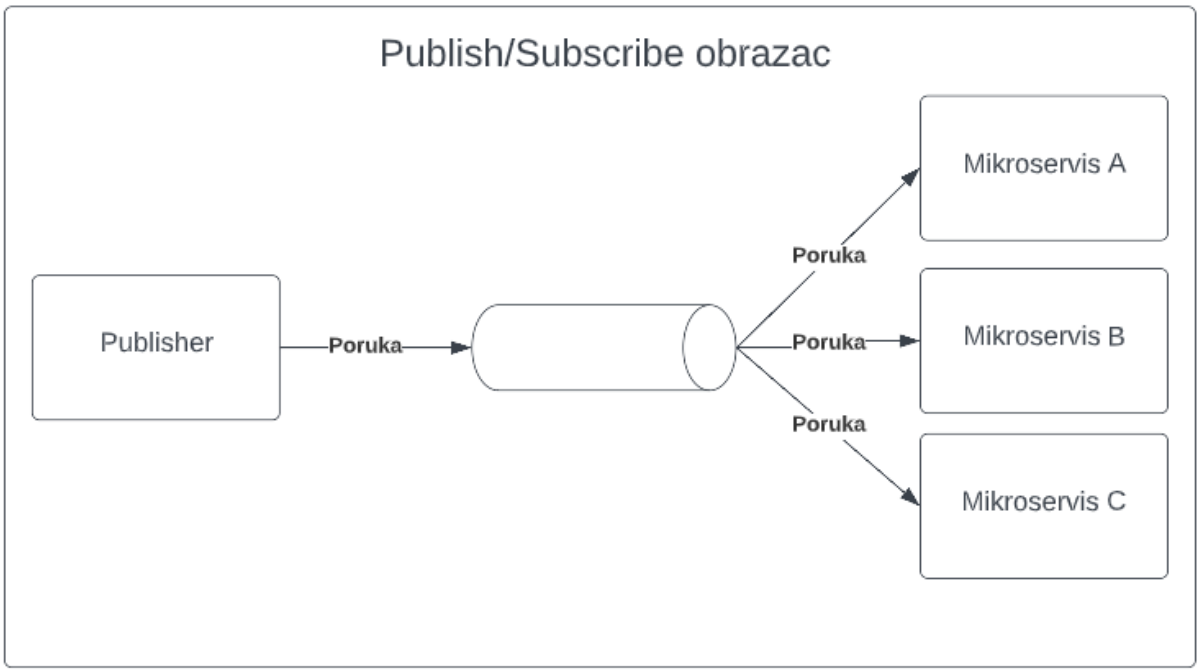
kada proizvođač šalje poruku. Ako stvaranje ili slanje poruke ne uspije, potrošač neće biti obaviješten i ne mora se brinuti o rješavanju takve greške. Umjesto toga, to je nešto za što je odgovoran proizvođač. S druge strane, ako konzumacija poruke ne uspije, proizvođač nije obaviješten i ne treba rješavati greške, već je odgovornost na potrošaču da riješi grešku.

3.4. Asinkroni komunikacijski obrasci

Asinkroni komunikacijski obrasci omogućuju servisima da se odvoje jedni od drugih kako bi se poboljšale performanse, skalabilnost i pouzdanost. U ovom odlomku ćemo objasniti najčešće korištene asinkrone komunikacijske obrasce, zajedno s razlozima kada i gdje ih koristimo.

3.4.1. Obrazac *publish/subscribe*

Obrazac *publish/subscribe* je oblik asinkronog slanja poruka gdje će proizvođač objaviti poruku na određenu temu (eng. *subject*), odakle će biti isporučena svim slušateljima pretplaćenim na tu temu [9]. To omogućuje da o porukama razmišljamo više kao događajima na koje mogu slušati različite vrste servisa. U praksi to čini vezu između izdavača i pretplatnika prilično labavom jer je samo vrsta izdavača obično specifična za temu, dok može postojati mnogo različitih vrsta pretplatnika. Sustavi koji omogućuju mehanizam *publish/subscribe* su općenito temeljeni na "guranju", što znači da je posrednik taj koji isporučuje poruke svakom pretplatniku, što se najčešće izvršava preko protokola *HTTP*. Mehanizam temeljen na "guranju" neće uspjeti ako pretplatnik nije dostupan, što znači da je potreban mehanizam ponovnog pokušaja (eng. *retry*) za upravljanje kratkotrajnim prekidom. Mehanizam mrtvog pisma (eng. *dead-letter*) koristan je za upravljanje duljim razdobljima nedostupnosti za pretplatnika ili trajno neuspješnom obradom uzrokovanom greškom u logici obrade pretplatnika [2]. Na slici *Slika **6*** prikazan je tradicionalan izgled *publish/subscribe* sustava, gdje vidimo kako tri potrošača sluša na kanalu od jednog izdavača.



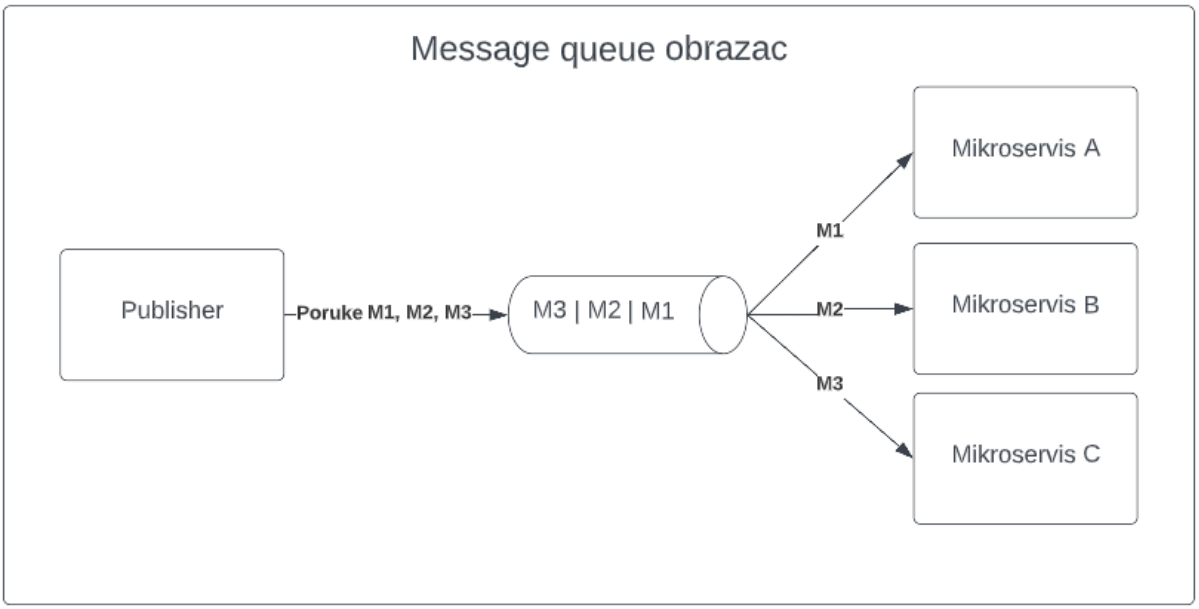
Slika 6: Obrazac publish/subscribe

Obrazac *publish/subscribe* se koristi kada se treba dogoditi više stvari za svaki pojedini događaj [9]. Na primjer, izvršenje jedne radnje rezultira time da je potrebno ažurirati više baza podataka, poslati obavijesti, ili pozvati daljnje servise. Navedeni obrazac nam također omogućuje dodavanje novih funkcionalnosti bez prekidanja ili mijenjanja postojećeg koda. Novi servisi mogu se jednostavno pretplatiti na kanal kako bi primali poruke i obrađivali iste.

3.4.2. Obrazac *message queue*

Message queue je asinkroni komunikacijski obrazac gdje proizvođač objavljuje poruke u red čekanja, iz kojeg će biti isporučena jednom od potrošača [9]. Dostava može biti temeljena na "povlačenju", pri čemu potrošači traže nove poruke od posrednika, ili na "guranju", kod koje posrednik isporučuje poruke potrošaču. To omogućuje učinkovito korištenje velikog broja instanci za obradu poruka iz reda čekanja. Pružatelji reda čekanja često pružaju neograničeno predmemoriranje (eng. *unbounded buffering*) za poruke, što povećava elastičnost skaliranja sustava budući da u velikom povećanju volumena, poruke neće ne biti isporučene i obrađene već će dostava i obrada biti odgođeni [2]. Općenito, redovi čekanja brišu poruku nakon što je isporučena i obrađena. Na slici *Slika **7*** prikazan

je tradicionalan izgled *message queue* sustava, gdje vidimo kako jedan izdavač šalje poruke u kanal na koji sluša tri različita mikroservisa (potrošača), te će svaki od njih obraditi jednu poruku.



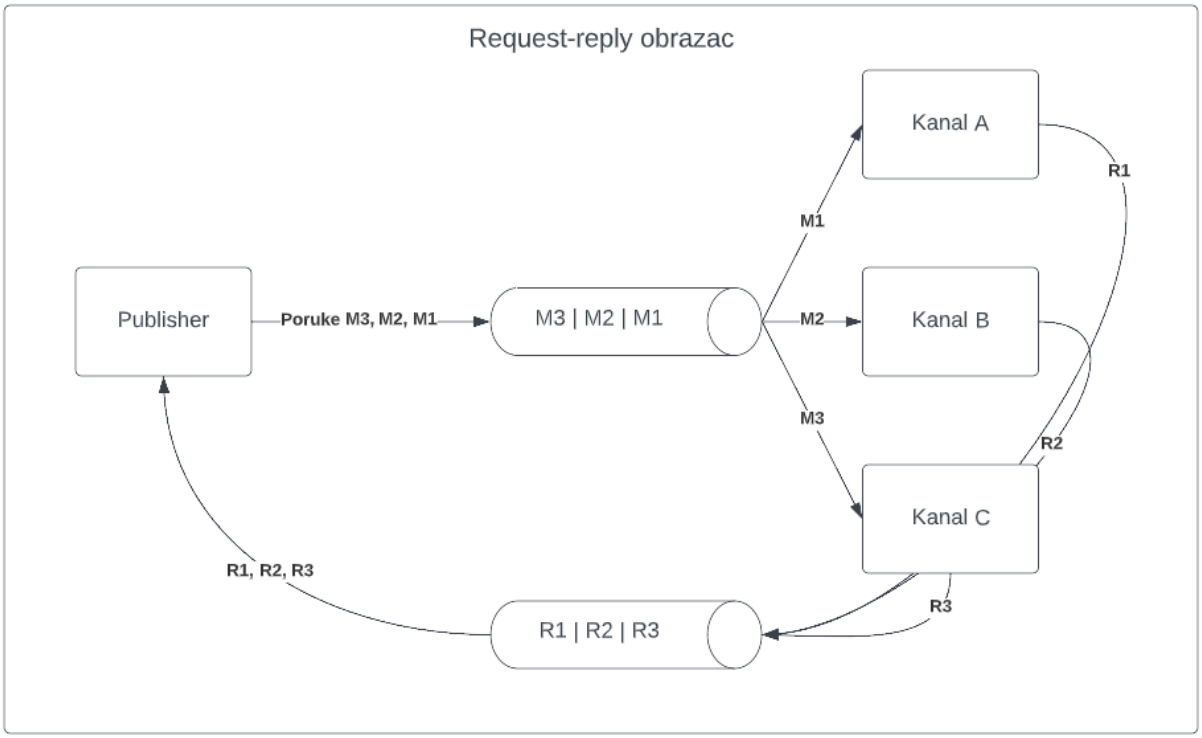
Slika 7: Obrazac message queue

Message queue sustavi *izvrsni* su za raspodjelu posla na više strojeva. Obično ti "radnički" strojevi imaju samo jedan posao, a to je obrada poruka iz reda čekanja. To ima mnoge prednosti kao što je uklanjanje opterećenja s klijenta, omogućavanje masivnog horizontalnog skaliranja i elegantno upravljanje sa skokovima opterećenja. Ako nema dovoljno strojeva za obradu svih poruka, poruke ostaju u redu dok ih korisnik ne obradi. Za skaliranje jednostavno dodajemo više radnih strojeva. Još jedna važna razliku između reda čekanja i *publish/subscribe* obrasca je da korisnik poruke iz reda čekanja mora potvrditi kako bi dao signal da je obrada te poruke dovršena. Ako korisnik ne potvrdi poruku nakon nekog vremena, servis reda čekanja mislit će da obrada nije uspjela i ponovno će staviti poruku u red čekanja kako bi je drugi korisnik mogao obraditi. Zbog toga je važno osigurati da je obrada idempotentna budući da je moguće obraditi poruku (ili djelomično obraditi poruku) više puta ako radni poslužitelj zakaže tijekom obrade [9].

3.4.3. Obrazac *request-reply*

Obrazac request-reply radi na jednostavnom principu gdje potrošač pošalje poruku, nakon čega zaprimi odgovor. Iako zvuči dosta poput sinkronog komunikacijskog obrasca kao što je *REST*, to nije potpuno istinito. Opća ideja je da proizvođač uključuje odredište za potrošača da objavi još jednu poruku putem *request-reply* obrasca [9]. To bi obično bio drugi kanal ili tema, ali može biti i *HTTP URL* ili nešto drugo, ovisno o sustavu.

Iako nije toliko uobičajen kao prethodno navedeni asinkroni komunikacijski obrasci, postoje neki slučajevi u kojima je koristan, kao što je kada čekanje odgovora može potrajati neko vrijeme ili ako je nepoznato tko će obraditi poruku. Na primjer, korisnik pošalje obrazac gdje obrada može potrajati nekoliko minuta. Poruka sadrži kanal na koji će preglednik slušati za odgovor. Kada proces u klasteru završi s obradom poruke, objavit će rezultate u poruci na kanalu za odgovor, preglednik će zaprimiti poruku o odgovoru i prikazati informacije korisniku, što se vidi na slici *Slika **8***.

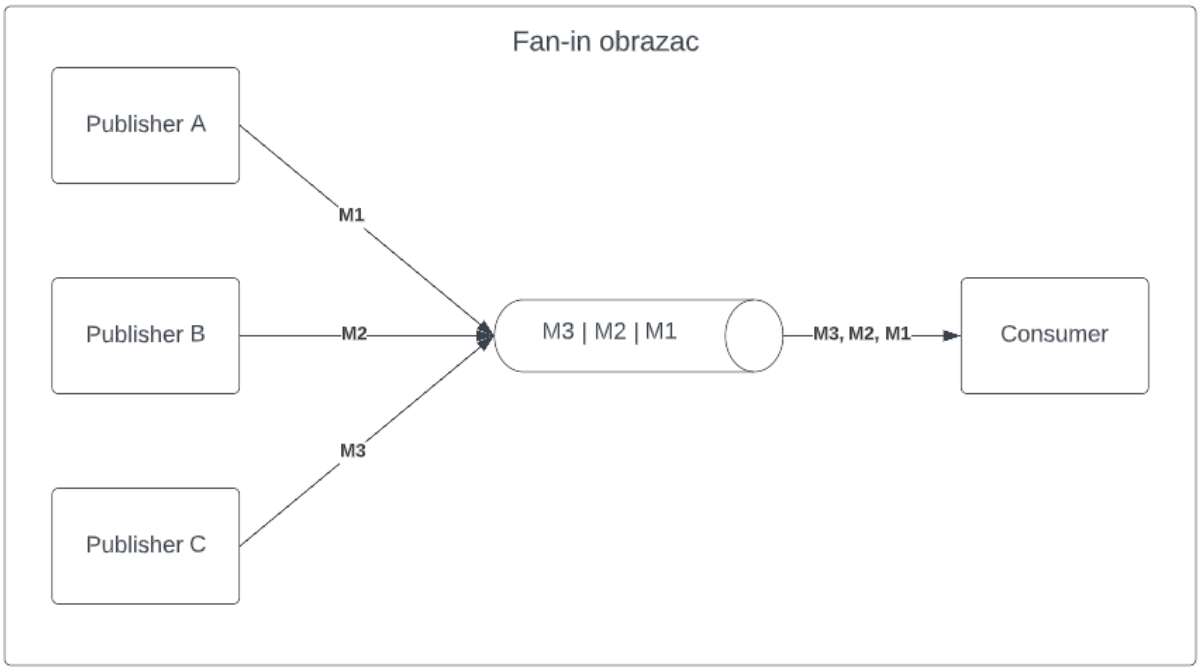


Slika 8: Obrazac request-reply

3.4.4. Obrazac *fan-in*

Obrazac *fan-in* se koristi kada moramo biti sigurni da samo jedan potrošač može primiti poruke na određenom kanalu [9]. Ovo nije vrlo uobičajen asinkroni komunikacijski obrazac jer postoji samo nekoliko slučajeva njegove upotrebe. Jedan od glavnih slučaja bi bio ako određeni potrošač mora pohraniti stanje između poruka ili ako se poruke moraju obrađivati određenim redoslijedom.

Obično je najbolje dizajnirati sustav na način gdje ovakav oblik komunikacije nije uvjetovan jer ozbiljno narušava sposobnost skaliranja. Također, nije dobro podržan sa strane posrednika. Na slici *Slika **9*** prikazan je jedan od mogućih izgleda *fan-in* obrasca, kod kojeg potrošač sluša na poruke od tri različita izdavača, i obrađuje svaku zasebno.



Slika 9: Obrazac fan-in

3.5. Usporedba sinkrone i asinkrone komunikacije

Sinkrona i asinkrona komunikacija su obje valjani načini komunikacije između servisa s njihovim prednostima i nedostatcima. Izbor za koje se odlučiti ovisi o potrebama nametnutim na sustav. Jednostavnost korištenja i integracije čini sinkronu komunikaciju valjanim izborom za mnoge slučajeve uporabe. Čak i unutar asinkrone arhitekture vođene događajima, sinkrona komunikacija može pronaći svoje mjesto u vanjskim integracijskim točkama i sučeljima okrenutim prema aplikacijama krajnjih korisnika [5]. Iako je *REST* današnji standard za razvoj aplikacijskog sučelja preko protokola *HTTP*, za bolje performanse možemo koristiti nešto poput *gRPC* programskog okvira i iskoristiti prednosti njegovog binarnog formata optimizirane veličine i poboljšane vrste komunikacije na strani klijenta, poslužitelja i dvosmjerne komunikacije [7]. Međutim, sinkrona komunikacija može biti više sklona kvarovima. Ako sinkroni servis naiđe na iznenadni porast zahtjeva i mehanizam automatskog skaliranja ne može pratiti brzinu povećanja, moguće je da servis bude preplavljen naglom potrebom za resurse. U najboljem slučaju to dovodi samo do povećanja vremena odgovora, ali također može rezultirati odbacivanjem zahtjeva ili rušenjem instanci servisa. Asinkrona komunikacija omogućuje otporniju i elastičniju arhitekturu s većom tolerancijom na greške. To postiže odvajanjem proizvođača i potrošača s posrednikom koji može spremati poruke u međuspremnik. Nedostatak asinkrone komunikacije je to što uvodi veću razinu složenosti na mnogo razina. Tijek komunikacije i obrade može završiti nepovezan na razini izvornog koda, što otežava praćenje samih tokova izvršenja. Osim toga, može zahtijevati puno dodatnih funkcija i sučelja za implementaciju zbog potrebe za dizajn i integraciju posrednika i baze podataka za trajno spremanje stanja i slanje i primanje poruka. Povećana je i infrastrukturna složenost. Posrednik je kritičan dio infrastrukture i stoga mora uvijek biti dostupan. U slučaju kvara, vrijeme oporavka i smanjenje gubitka podataka je kritično. Postizanje potrebne razine servisa za posrednika može uključivati puno posla samo po sebi i budući da je to sustav koji neprekidno radi, održavanje i ažuriranja su složenija. Količina infrastrukturnih radova ovisi o sustavu.

4. Upravljanje složenim komunikacijskim procesima unutar raspodijeljenog sustava

API Gateway i *service mesh* postali su poznati kao načini upravljanja složenim komunikacijskim procesima aplikacija koji se odvijaju unutar raspodijeljenog sustava [10]. Obje, na svoj način, igraju ulogu u podržavanju aplikacija visokih performansi koje su skalabilne koliko god je to moguće. Arhitekti i programeri ih koriste za pojednostavljenje komunikacije između različitih softverskih resursa, izlaganje podataka potrebnih za nadziranje i promatranje transakcija unutar aplikacija, i eliminaciju potrebe za implementiranjem komunikacijske logike izravno u pojedinačan servis ili aplikaciju [11].

4.1. Infrastrukturni sloj *service mesh*

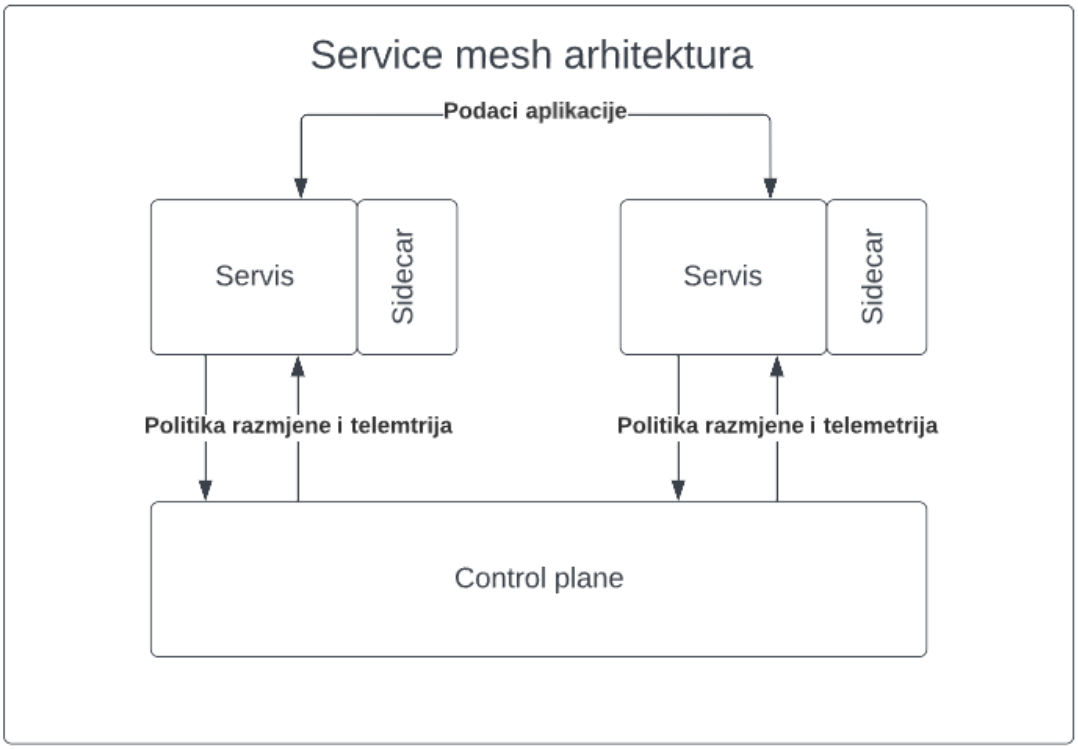
Mikroservisi i raspodijeljene aplikacije zahtijevaju brzu mrežnu komunikaciju i složene orkestracije kako bi se osigurale visoke performanse i brza skalabilnost. Razine zamršenosti zahtijevaju potpunu automatizaciju upravljanja, budući da složenost nadilazi mogućnost *IT* administratora da ručno definira kontrole, kao što su *load balancing*, pravila usmjeravanja (eng. *routing rules*), *service discovery* i autentifikacija [12]. U te svrhe možemo koristiti *service mesh*, koji predstavlja alat za upravljanje komunikacijskim procesima koji se odvijaju unutar raspodijeljenog sustava [12].

Service mesh je namjenski infrastrukturni sloj koji kontrolira komunikaciju između servisa preko mreže. Ova metoda omogućuje međusobnu komunikaciju zasebnih dijelova aplikacije. *Service mesh* obično se pojavljuje kod aplikacija temeljenih na oblaku, spremnika i mikroservisa.

Service mesh kontrolira isporuku zahtjeva u aplikaciji. Uobičajene značajke koje *service mesh* pruža uključuju *service discovery*, *load balancing*, šifriranje i oporavak od kvara. *Service mesh-evi* mogu učiniti komunikaciju između servisa brzom, pouzdanom, sigurnom, i visoko dostupnom. Organizacije mogu odabrati *API Gateway*, koji upravlja transakcijama protokola, umjesto *service mesh-a*. Međutim, programeri moraju ažurirati *API Gateway* svaki put kada se stvori ili ukloni mikroservis. *Service mesh* obično nudi

skalabilnost i fleksibilnost kod upravljanja mrežom koja nadilazi mogućnosti tradicionalnih *API Gateway-a* [12].

Na slici *Slika **10*** prikazana je arhitektura *service mesh-a* koja koristi instancu posrednika (engl. *proxy*) koja se naziva *sidecar* u bilo kojoj razvojnoj paradigmi koja se koristi, obično spremnicima ili mikroservisima [12]. U mikroservisnoj aplikaciji, *sidecar* se pridružuje svakom pojedinačnom servisu. U spremniku, *sidecar* se pričvršćuje na svaki spremnik aplikacije, virtualnu mašinu, ili jedinicu za orkestraciju spremnika, kao što je *Kubernetes pod*. *Sidecar* servisi mogu također rješavati zadatke koji su apstrahirani od samih servisa, kao što su nadziranje i sigurnost [10]. Instance servisa, *sidecar* servisi i njihove interakcije čine ono što se naziva podatkovna ravnina u arhitekturi *service mesh-a* [12]. Drugi sloj koji se naziva upravljačka ravnina upravlja zadacima kao što su stvaranje instanci, nadziranje i implementacija politika za upravljanje mrežom i sigurnosti. Upravljačke ravnine se mogu spojiti na *CLI* (eng. *Command Line Interface*) ili *GUI* (eng. *Graphical User Interface*) sučelje za upravljanje aplikacijama. *Sidecar* je instanca posrednika koja prati *ambassador* obrazac.



Slika 10: Arhitektura service mesh-a

Aplikacija strukturirana kao mikroservisna arhitektura može sadržavati desetke ili stotine servisa, sve sa svojim vlastitim instancama koje rade u produkcijskom okruženju. Za programere je veliki izazov pratiti koje komponente moraju međusobno komunicirati, nadzirati njihovo zdravlje, performanse i ažurirati servis ili komponentu ako nešto pođe po zlu. *Service mesh* omogućuje programerima odvajanje i upravljanje komunikacijama između servisa u namjenskom infrastrukturnom sloju [12]. Kako se broj mikroservisa uključenih u aplikaciju povećava, tako rastu i prednosti korištenja *service mesh-a* za njihovo upravljanje i nadziranje.[1]

Service mesh obično pruža mnoge mogućnosti koje komunikaciju u spremnicima i mikroservisima čine pouzdanijom, sigurnijom i vidljivom. Ključne značajke *service mesh-a* navedene su ispod.[2][3]

Usmjeravanje prometa. *Service mesh* može usmjeravati zahtjeve prema instanci servisa na temelju pravila ili konfiguracije [12]. Prometu iz klijentske aplikacije može se dati prioritet ili se promet može selektivno usmjeriti na drugu verziju servisa kao podrška za:

- *Canary* distribucije
- *A/B* testiranje
- Upravljanje verzijama

Pouzdanost. Upravljanje komunikacijama putem *sidecar* posrednika i upravljačke ravnine poboljšava učinkovitosti pouzdanost zahtjeva servisa, politika i konfiguracija. Specifične mogućnosti uključuju *load balancing* i ubacivanje grešaka [12].

Preglednost. *Service mesh* može pružiti uvid u ponašanje i zdravlje servisa. Upravljačka ravnina može prikupiti i agregirati telemetrijske podatke iz interakcija komponenti kako bi se utvrdilo zdravlje servisa, kao što su promet i latencije, raspodijeljeno praćenje i bilješke pristupa [12]. Integracija alata treće strane, kao što su *Prometheus*, *Elasticsearch* i *Grafana*, omogućuje daljnje praćenje i vizualizaciju.

[1] Prometheus – sustav za nadziranje
[2] Elasticsearch – raspodijeljeni sustav koji pruža mogućnost otvorenog traženja i služi kao analitički mehanizam za mnoge tipove podataka (tekstualne, numeričke, geoprostorne, strukturirane)
[3] Grafana – sustav za analitiku, nadziranje i interaktivnu vizualizaciju podataka

Sigurnost. *Service mesh* može automatski šifrirati komunikaciju i distribuirati sigurnosne politike, uključujući autentifikaciju i autorizaciju, od mreže do aplikacije i pojedinačnih mikroservisa [12]. Centralno upravljanje sigurnosnim politikama putem upravljačke ravnine i *sidecar* posrednika pomaže u održavanju koraka sa sve složenijim vezama unutar i između raspodijeljenih aplikacija.

Elastičnost. Politike *retry* koje provodi posrednik potpuno izoliraju programere od scenarija u kojima je pozvani servis nakratko nedostupan. Posrednik bi također mogao pokušati alternativni put do servisa ili prijeći na rezervni servis. Programeri mogu vjerovati da je posrednik dao sve od sebe da riješi komunikacijske pogreške ako poziv prema servisu ne uspije [12]. Osim toga, mreža može pružiti mogućnost usmjeravanja do instance servisa s najnižom latencijom za optimalne performanse. Primjeri obrazaca otpornosti koje možemo konfigurirati i nametnuti su:

- Politike *retry*
- Obrazac *circuit breaker*
- Mehanizmi ograničavanja stope (eng. *rate limiting*) i prigušivanje (eng. *throttling*)

Service mesh rješava neke velike probleme s upravljanjem komunikacijom između servisa, ali ne sve. Jedna od prednosti je što pojednostavljuje komunikaciju između servisa u mikroservisima i spremnicima. Također, jednostavnije je dijagnosticirati komunikacijske greške, jer se pojavljuju na vlastitoj infrastrukturnoj razini. Podržava sigurnosne značajke kao što su šifriranje, autentifikacija i autorizacija te omogućuje brži razvoj, testiranje i distribuciju aplikacije [12]. Posljednje, *sidecar* servisi postavljeni pored klastera spremnika je učinkovit način upravljanja mrežnim servisima.

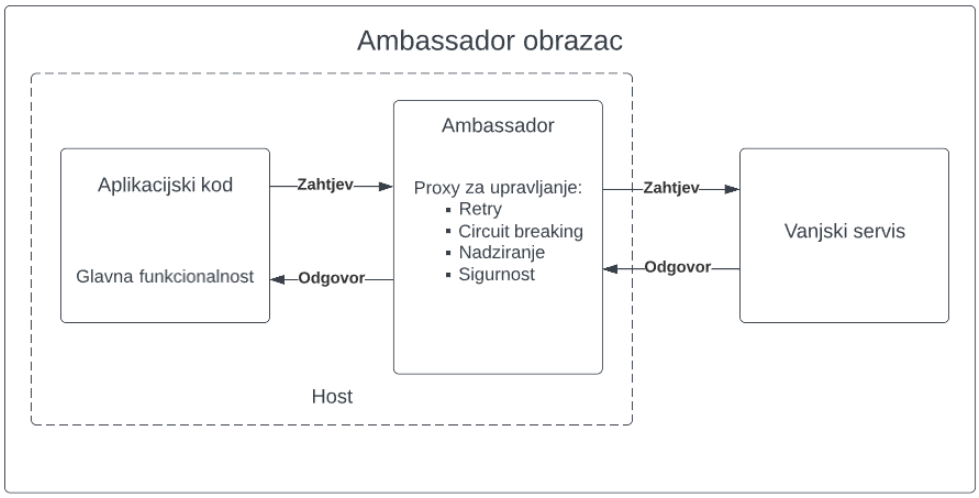
Neki od nedostataka *service mesh-a* su što se povećava broj instanci tijekom izvođenja programa. Uz navedeno, svaki poziv prema servisu prvo mora proći kroz *sidecar* posrednika, što nadodaje jedan korak. Iako je složenost upravljanja mrežom apstrahirana i centralizirana, nije eliminirana. Netko mora integrirati *service mesh* u radno okruženje i upravljati njenom konfiguracijom.

4.1.1. Obrazac *ambassador*

Obrazac *ambassador* stvara pomoćne servise koji šalju mrežne zahtjeve u ime potrošačkog servisa ili aplikacije. *Ambassador servis* može se smatrati izvan procesnim posrednikom koji dijelu lokaciju klijenta [13]. Ovaj obrazac može biti koristan za rasterećenje uobičajenih zadataka povezivanja klijenta kao što su nadziranje, bilježenje, usmjeravanje, sigurnost i obrasci otpornosti (eng. *resiliency*) na jezično agnostički način [13]. Često se koristi s nasljednim aplikacijama, ili drugim aplikacijama koje je teško modificirati kako bi se proširile njihove mrežne mogućnosti. Također može omogućiti specijaliziranom timu implementaciju tih značajki.

Otporne aplikacije temeljene na oblaku zahtijevaju implementaciju značajki kao što su *circuit breaker*, usmjeravanje, mjerenje i nadziranje te mogućnost ažurirana konfiguracije vezane uz mrežu. Moguće je da će biti teško ili nemoguće ažurirati nasljedne aplikacije ili postojeće biblioteke koda kako bi se dodale navedene značajke jer se kod više ne održava ili ga razvojni tim ne može lako modificirati. Mrežni pozivi također mogu zahtijevati značajnu konfiguraciju za povezivanje, provjeru autentičnosti i autorizaciju. Ako se ti pozivi koriste u više aplikacija, izrađenih pomoću više programskih jezika i programskih okvira, pozivi moraju biti konfigurirani za svaku od tih instanci. Osim toga, mrežnom i sigurnosnom funkcionalnosti možda će morati upravljati centralni tim unutar organizacije. S velikom bazom koda, za taj tim može biti rizično ažurirati kod aplikacije s kojim nisu upoznati.

Kao rješenje možemo staviti klijentske programske okvira i biblioteke u vanjski proces koji djeluje kao posrednik između naše aplikacije i eksternih servisa [13]. Možemo postaviti posrednika u okruženje naše aplikacije kako bismo omogućili kontrolu nad usmjeravanjem, otpornošću, sigurnosnim značajkama, i kako bi izbjegli bilo kakva ograničenja pristupa povezanih s aplikacijom, što se vidi na slici *Slika **11***. Također, možemo koristiti *ambassador* obrazac za standardizaciju i proširenje instrumentacije [13]. Posrednik može nadzirati metrike performansi kao što su latencije ili iskorištenje resursa, te se to nadziranje odvija u istom okruženju kao i aplikacija.



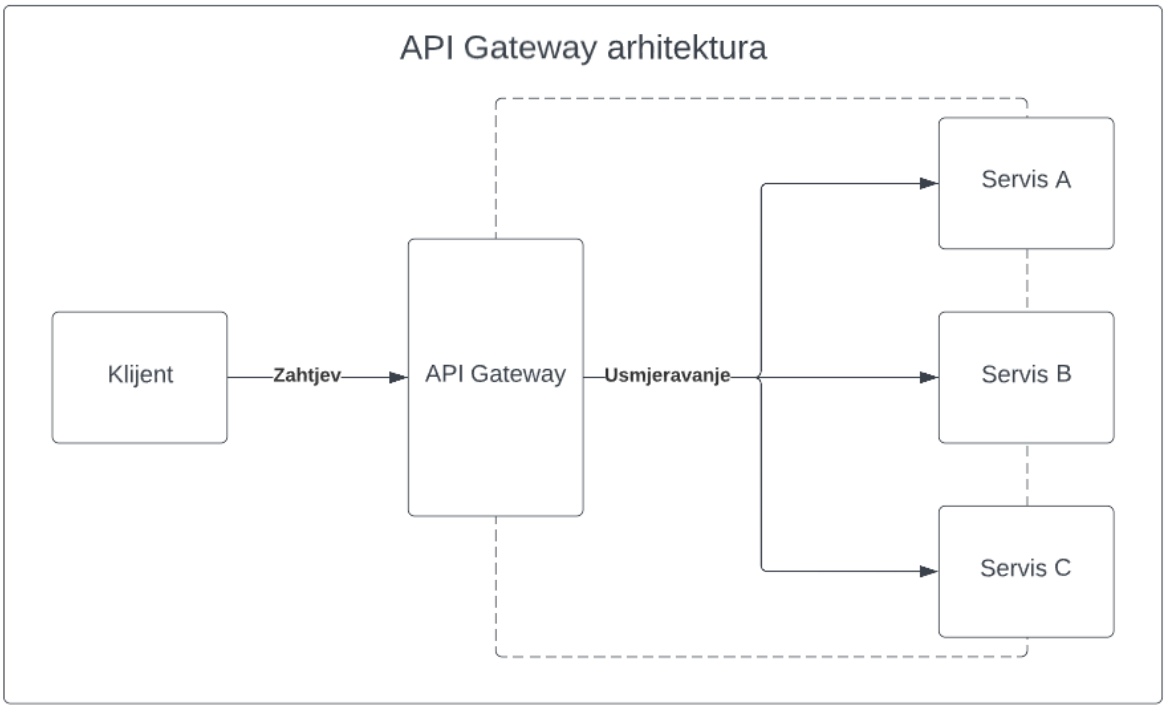
Slika 11: Obrazac ambassador

Značajke koje se prenose na *ambassador-a* mogu se upravljati neovisno o aplikaciji. Možemo ažurirati i modificirati *ambassador-a* bez ometanja naslijeđene funkcionalnosti aplikacije. Također, omogućuje zasebnim, specijaliziranim timovima da implementiraju i održavaju sigurnosne i mrežne značajke, ili provjeru autentičnosti koje su premještene na *ambassador-a*. *Ambassador* servisi mogu se postaviti kao *sidecar* servisi koji prate životni ciklus aplikacije ili servisa koji se koristi. Ako se potrošački servis izvršava unutar spremnika, *ambassador* treba biti stvoren kao zaseban spremnik na istoj *host* mašini, s odgovarajućim vezama konfiguriranim za komunikaciju [13].

4.2. Alat za upravljanje aplikacijskim sučeljima – *API Gateway*

API Gateway je softverski alat za upravljanje sustavima aplikacijskih sučelja koji se nalazi ispred aplikacijskih sučelja ili grupe mikroservisa, kako bi se olakšali zahtjevi i isporuka podataka i servisa [14]. Na slici *Slika **12***, vidimo da je njegova primarna uloga djelovati kao jedinstvena ulazna točka i standardizirani proces za interakcije između aplikacija, podataka i servisa organizacije te unutarnjih i vanjskih korisnika [15]. Također *API Gateway* može obavljati razne druge funkcije za podršku i upravljanje sustavima aplikacijskih sučelja, od provjere autentičnosti do ograničavanja stope i analitike. Možemo reći da *API Gateway* djeluje kao obrnuti posrednik (eng. *reverse proxy*) za prihvaćanje svih poziva aplikacijskih sučelja, agregiranje različitih servisa potrebnih za

njihovo ispunjavanje, i vraćanje odgovarajućeg rezultata [14]. Oni su ključni kod mikroservisne arhitekture, u kojoj zahtjevi za podacima pozivaju brojne aplikacije i servise koji koriste više različitih sustava aplikacijskih sučelja. Uloga *API Gateway-a* u takvom slučaju je omogućiti jedinstvenu točku ulaza za definiranu grupu mikroservisa i primijeniti pravila za određivanje njihove dostupnosti i ponašanja. Često obrađuju i druge funkcije povezane sa mikroservisnom arhitekturom kao što su *service discovery*, autentifikacija i provedba sigurnosne politike, *load balancing*, upravljanje privremenom memorijom (eng. *caching*), nadziranje, bilježenje i analitike [10].



Slika 12: Arhitektura API Gateway-a

Primarna prednost *API Gateway-a* je što standardizira i centralizira isporuku servisa putem *API* sustava ili mikroservisa. Osim toga, *API Gateway* pomaže osigurati i organizirati integracije temeljenje na sustavu aplikacijskog sučelja na brojne načine.

Pojednostavljuje isporuku servisa. *API Gateway* može kombinirati višestruke pozive aplikacijskog sučelja za traženje i dohvaćanje podataka i servisa, što smanjuje količinu zahtjeva i prometa. To omogućuje procesu aplikacijskog sučelja da bude tečan što može poboljšati korisničko iskustvo [11].

Fleksibilnost. *API Gateway* je visoko podesiv. Programeri mogu enkapsulirati unutarnju strukturu aplikacije na više načina, kako bi pozvali više servisa i agregirali rezultate [10].

Proširenje nasljednih aplikacija. Organizacije koje se oslanjaju na nasljedne aplikacije mogu koristiti *API Gateway* za rad s tim aplikacijama i čak proširiti njihovu funkcionalnost, kao alternativa široj i kompliciranoj migraciji [10].

Nadziranje i preglednost. Većina organizacija oslanja se na specifične alate za nadziranje aktivnosti sustava aplikacijskog sučelja, ali *API Gateway* može pomoći u tim naporima [14]. Bilješke koje proizvodi *API Gateway* mogu pomoći u otkrivanju problema tijekom neuspjelog događaja nadziranja.

S obzirom na to da je *API Gateway* posrednik između potrošača i pružatelja aplikacijskog sučelja, ta široka uloga predstavlja i mnoge izazove.

Pouzdanost i otpornost. Svako oštećenje ili smetnja funkcionalnosti *API Gateway* može uzrokovati kvar povezanih servisa. Organizacije moraju biti oprezne pri dodavanju značajki koje utječu na performanse, posebno zato što *API Gateway* predstavlja dodatni korak u procesu između korisnika i aplikacija ili podataka [10].

Sigurnost. *API Gateway* pouzdan je izvor koji dotiče mnoge kutove poslovanja poduzeća. Ako je ugrožen, to je potencijalno ozbiljan i dalekosežan sigurnosni problem [15]. Poduzeća bi trebala pažljivo odvojiti vanjska sučelja od internih sustava aplikacijskih sučelja te definirati parametre provjere autentičnosti i autorizacije.

Složenost i ovisnosti. Programeri moraju ažurirati *API Gateway* svaki put kada se aplikacijsko sučelje ili mikroservis stvori, promjeni ili izbriše [14]. To je posebno izazovno u modelu u kojem bi nekoliko aplikacija moglo postati deseci ili stotine mikroservisa. Stvaranje i pridržavanje dizajna sustava aplikacijskih sučelja i mikroservisa može pomoći u ublažavanju ovih problema.

4.3. Obrasci otpornosti za bolju pouzdanost

Otpornost aplikacije je njezina sposobnost oporavka od kvarova. Prilikom izgradnje mikroservisa, važno je razmotriti koliko su ti brojni raspodijeljeni servisi otporni. Aplikacije temeljene na mikroservisnoj arhitekturi često imaju nekoliko ovisnosti, uključujući baze podataka, komponente pozadinskog sustava i sustave aplikacijskih sučelja koji potencijalno mogu uzrokovati neuspjehe kod poziva servisa [16]. Takva interakcija čini stabilnost jednog mikroservisnog sustava ili servisa uvelike ovisnom o dostupnosti i opterećenosti drugih sustava ili servisa. Mreže su nepouzdane, a vršna opterećenja jedan su od glavnih uzroka kvarova. U ovom poglavlju ćemo predstaviti nekoliko obrasca koji rješavaju taj problem. Neki su proaktivni, pomažu u sprječavanju kvarova, dok su drugi reaktivni, upravljaju kvarovima u trenutku kada se pojave [17].

Obrazac otpornosti	***Reaktivan/proaktivan***
Retry	Reaktivan
Timeout	Reaktivan
Fallback	Reaktivan
Cache	Reaktivan/Proaktivan
Rate limiter	Proaktivan
Bulkhead	Proaktivan
Circuit breaker	Reaktivan

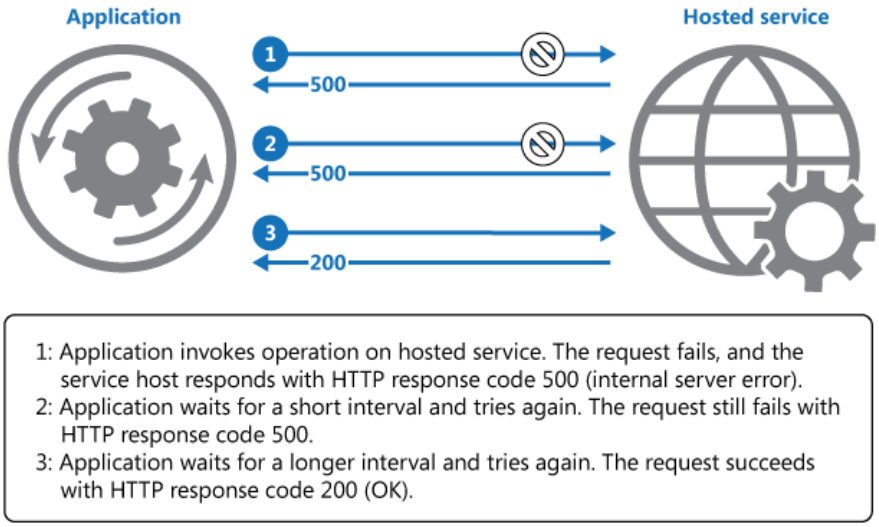
Tablica 8: Obrasci otpornosti

4.3.1. Obrazac *retry*

Mikroservisi često imaju mnoge ovisnosti, uključujući baze podataka, komponente pozadinskih sustava i sustave aplikacijskih sučelja. Bilo koja od ovih ovisnosti može povremeno otkazati i posljedično stvoriti brojne neuspjehe kod poziva servisa. Obrazac ponovnog pokušaja pruža rješenje za ove tranzijentne pogreške. Tijekom povremenih i trenutnih kvarova, obrazac ponovnog pokušaja stvara mehanizam koji ponavlja neuspjelu operaciju određeni broj puta [18]. *IT* administratori mogu konfigurirati određeni broj ponovnih pokušaja i vremenske intervale između njih. Ovo pruža

neuspjelim servisima priliku za pozivanje servisa jednom ili više puta dok se ne primi očekivani odgovor od servisa, umjesto da se jednostavno isključi nakon početnog kvara. Važno je izbjegavati ulančavanje ponovnih pokušaja i koristiti ovaj obrazac samo za tranzijentne kvarove. Također, korisno je održavati bilješke kako bismo kasnije utvrdili glavni uzrok takvih kvarova. Konačno, svakom servisu je potrebno dati dovoljno vremena za oporavak kako bismo spriječili kaskadne kvarove i sačuvali mrežne resurse dok se neuspjeli servis ne oporavi.

Dobra strategija za ponovni pokušaj je korištenje eksponencijalnog odmaka, pri čemu se vrijeme između ponovnih pokušaja povećava nakon svakog pokušaja [18]. Tako će se prvi ponovni pokušaj provesti nakon jedne sekunde, sljedeći nakon dvije, itd. Najbolje je ograničiti interval ponovnog pokušaja na neku maksimalnu vrijednost, inače bi se vrijeme obrade sustava moglo previše povećati. Na slici *Slika* ***13*** prikazana je ilustraciju pozivanja servisa korištenjem *retry* obrasca.

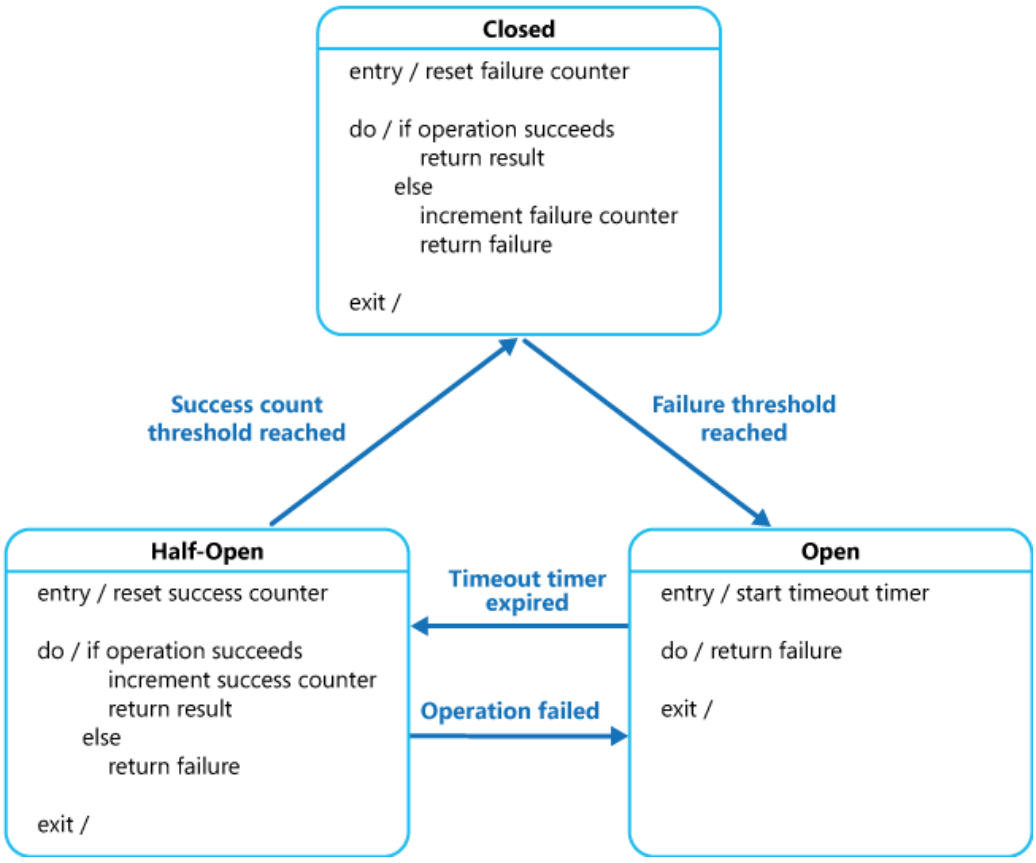


Slika 13: Ilustracija retry mehanizma [18]

4.3.2. Obrazac *circuit breaker*

Iako obrazac ponovnog pokušaja funkcionira kod tranzijentnih kvarova, timovi i dalje trebaju pouzdan obrazac otpornosti koji obrađuje veće, dugoročne, trajne kvarove. Ako mehanizam ponovnog pokušaja poziva ozbiljno oštećen servis nekoliko puta dok ne dobije željeni rezultat, to bi moglo rezultirati kaskadnim kvarovima servisa koje postaje sve teže identificirati i popraviti. Obrazac *circuit breaker* stvara komponentu koja nalikuje

tradicionalnom električnom prekidaču [19]. Ta komponenta se nalazi između servisa koji iniciraju zahtjev i krajnjih servisa koji izlažu aplikacijsko sučelje. Sve dok ti servisi normalno komuniciraju, *circuit breaker* delegira poruke između njih u zatvorenom stanju. Kada ponovljeni zahtjev servisa koji putuje kroz zatvoreni krug ne uspije unaprijed određeni broj puta, *circuit breaker* prelazi u otvoreno stanje kako bi zaustavio izvršenje servisa. Tijekom ovog otvorenog stanja, *circuit breaker* zaustavlja izvršenje servisa i vraća poruke o pogreški servisu koji je inicirao zahtjev za svaku neuspjelu transakciju. Nakon određenog vremenskog intervala (eng. *circuit reset timeout*), *circuit breaker* radi u poluotvorenom stanju [19]. Tijekom tog vremena, *circuit breaker* zatvara petlju kako bi provjerio je li veza između dva servisa ponovno uspostavljena. Ako *circuit breaker* i dalje otkrije jednu grešku, ponovno će se aktivirati u otvoreno stanje. Nakon što se pogreška riješi, ponovno se zatvara petlja kao i obično. Ilustraciju navedenog mehanizma vidimo na slici *Slika* ***14***.



Slika 14: Ilustracija circuit breaker mehanizma [19]

4.3.3. Obrazac *rate limiter*

Obrazac *rate limiter* nam pomaže da naše servise učinimo visoko dostupnima samo ograničavanjem broja poziva koje možemo uputiti određenom servisu [20]. Drugim riječima, pomaže nam kontrolirati propusnost. Vršna opterećenja i nesputano korištenje resursa glavni su uzrok kvarova sustava. *Rate limiting* je proaktivna strategija koja pomaže u sprječavanju toga. Kada naš sustav doživi vršno opterećenje zbog mnogih dolaznih zahtjeva, ako uputimo mnogo odlaznih poziva prema eksternom sustavu, možemo uzrokovati vršno opterećenje i na tom sustavu, čak ga preopteretiti i srušiti. S *rate limiter* obrascem možemo rasporediti ta vršna opterećenja tako da dozvolimo obradu određenog broja zahtjeva u određenom intervalu, ili možemo napraviti samo jedan poziv za učestale iste zahtjeve i spremiti odgovor u privremenu memoriju kako bismo naše servise učinili visoko dostupnim [20].

```go
package main

import (
    "fmt"
    "time"
)

func main() {

    requests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        requests <- i
    }
    close(requests)

    limiter := time.Tick(1000 * time.Millisecond)

    for req := range requests {
        <-limiter
        fmt.Println("request", req, time.Now())
    }

    burstyLimiter := make(chan time.Time, 3)

    for i := 0; i < 3; i++ {
        burstyLimiter <- time.Now()
    }

    go func() {
        for t := range time.Tick(1000 * time.Millisecond) {
            burstyLimiter <- t
        }
    }()

    burstyRequests := make(chan int, 5)
    for i := 1; i <= 5; i++ {
        burstyRequests <- i
    }
    close(burstyRequests)

    for req := range burstyRequests {
        <-burstyLimiter
        fmt.Println("request", req, time.Now())
    }
}
```

Slika 15: Implementacija rate limiter obrasca [21]

4.4. Usporedba *API Gateway* i *service mesh* sustava

Eksterna vs. interna komunikacija. *API Gateway* upravlja zahtjevima koji potječu izvana, kao što je zahtjev korisnika aplikacije za pregled određene stranice. Nasuprot tome, *service mesh* obrađuje interne zahtjeve koje mikroservisi upućuju drugim mikroservisima unutar aplikacije. U tehničkom smislu, *API Gateway* nadzire komunikaciju između klijenta i poslužitelja, dok se *service mesh* bavi komunikacijom između internih servisa [10].

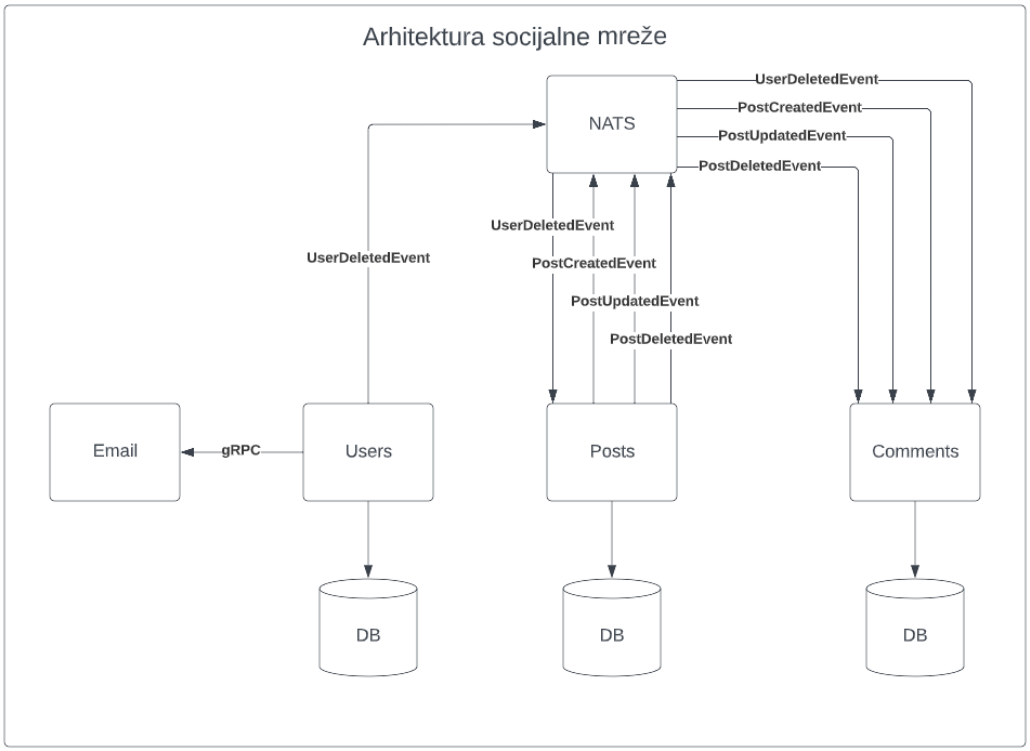
Smještaj unutar arhitekture. *API Gateway* i *service mesh* rade unutar različitih slojeva tipične softverske arhitekture. *API Gateway* je obično javna infrastrukturna komponenta, smještena između ruba mreže i pozadinskog dijela aplikacije, uključujući *service mesh* [10]. Kako vanjske komponente i sustavi šalju zahtjeve prema aplikaciji, te zahtjeve prima i potvrđuje *API Gateway*. Na kraju će *service mesh* vjerojatno pomoći u provođenju zahtjeva koje *API Gateway* odobri i proslijediti ih prema sustavu aplikacijskog sučelja.

Distribucija i upravljanje. U usporedbi sa *service mesh-om*, *API Gateway* je često jednostavniji za upravljanje. Obično samo jednom moramo postaviti *API Gateway* unutar softverske arhitekture, a oni su obično pogodni za jednostavno, centralizirano nadziranje. *Service mesh* se treba integrirati i prilagoditi svakom aplikacijskom servisu kojem pomaže kod upravljanja. Najčešći pristup implementaciji je pokretanje agenata *service mesh-a* u posredničkom spremniku koji se često naziva *sidecar* [10].

Nadziranje i preglednost. Metrike *API Gateway-a* pružaju najveću vrijednost pri praćenju ukupnog stanja aplikacije, kao što je koliko je vremena potrebno da se odgovori na određene zahtjeve aplikacijskog sučelja ili koliko veliki promet utječe na performanse [10]. Te metrike mogu pomoći u prepoznavanju problema s određenim sustavima aplikacijskih sučelja, osobito kada se ti problemi ne mogu pripisati ispadu mreže, padu aplikacije ili nekom drugom uobičajenom uzroku. U međuvremenu, metrike *service mesh-a* pomažu timovima identificirati probleme s pojedinačnim mikroservisima i komponentama koje nastanjuju pozadinski sustav aplikacije, a ne cjelokupnu aplikaciju [10]. Nadziranje koje pruža je korisno za precizno određivanje izvora problema s performansama određenih aplikacija. Međutim, vjerojatno neće pružiti nikakve naznake učinka tih problema iz vanjske perspektive.

5. Studija slučaja: Sinkroni i asinkroni komunikacijski obrasci unutar *kind* razvojnog okruženja

Kao sustav za razmjenu poruka koji će nam omogućiti asinkronu komunikaciju između servisa unutar *kind* razvojnog okruženja odabrali smo *NATS*. *NATS* će nam služiti kao vezivno tkivo koje će primati poruke i prosljeđivati ih servisima koji slušaju na te vrste poruka [22]. Uz *NATS*, određenim servisima je omogućena i međusobna komunikacija putem sinkronih komunikacijskih obrasca, specifično *REST* i *gRPC* programskog okvira. U svrhe ove studije slučaja, implementirali smo pet vrsta servisa čija je svrha simulirati funkcionalnosti jedne jednostavne socijalne mreže, što možemo vidjeti na slici *Slika **16***. Neke od bitnijih funkcionalnosti uključuju stvaranje i upravljanje korisnicima, objavama, komentarima na objave, slanjem elektroničke pošte na korisnikovu adresu i međusobno obavještavanje o određenim događajima kako bi servisi mogli pravilno funkcionirati.



Slika 16: Arhitektura socijalne mreže

5.1. Razvojno okruženje *kind*[4][5]

Kao razvojno okruženje za implemen[6]taciju socijalne mreže koristili smo *kind*. Alat *kind* se koristi za pokretanje *Kubernetes* klastera korištenjem *Docker* spremnika zvanih *nodes* [23]. Iako je primarno dizajniran za testiranje samog *Kubernetes* sustava, često se koristi kao lokalno razvojno okruženje ili za kontinuiranu integraciju i isporuku. Sastoji se od paketa implementiranih u programskom jeziku *Golang* koji implementiraju funkcionalnosti kao što su stvaranje klastera i izgradnja *Docker* slika [23]. Uz navedene pakete, sadržava i komandno sučelje izgrađeno nad tim paketima kako bi mogli upravljati našim razvojnim okruženjem. Kako bi si pojednostavili konstantno ponavljanje istih komandi za upravljanje našim *kind* razvojnim okruženjem, koristimo *make* alat za automatizaciju dugih naredbi. Stoga, naš *kind* klaster pokrećemo naredbom *make kind-up*, što pokreće isječak koda koji se nalazi ispod.

```
SHELL := /bin/bash

KIND_CLUSTER := social-network-cluster

kind-up:
    kind create cluster \
        --image
kindest/node:v1.21.1@sha256:fae9a58f17f18f06aeac9772ca8b5ac680ebbed985e266f711
d936e91d113bad \
        --name $(KIND_CLUSTER) \
        --config zarf/k8s/kind/kind-config.yaml
    kubectl config set-context --current --namespace=services-system
```

Slika 17: Instrukcija za pokretanje lokalnog kind klastera

Iz prethodne komande možemo vidjeti da *kind* učitava konfiguraciju iz *kind-config.yaml* datoteke, koja je prikazana na slici *Slika **18***. U našem slučaju, jedina vrsta konfiguracije koja nam je potrebna je izlaganje *port* točaka kako bi mogli komunicirati sa servisima unutar našeg klastera. Zbog činjenice da se naš klaster inicijalizira kao *Docker* spremnik,

[4] kind – alat za pokretanje lokalnog Kubernetes klastera
[5] Docker – ekosustav sagrađen oko ideje pokretanja softvera u spremnicima
[6] Kubernetes – orkestralni sustav za upravljanje, automatizaciju distribucije i skaliranje aplikacija koje se pokreću u spremnicima

imamo pristup samo jednoj mreži, stoga, moramo izložiti zaseban *port* za svaki servis koji pokrećemo unutar našeg klastera.

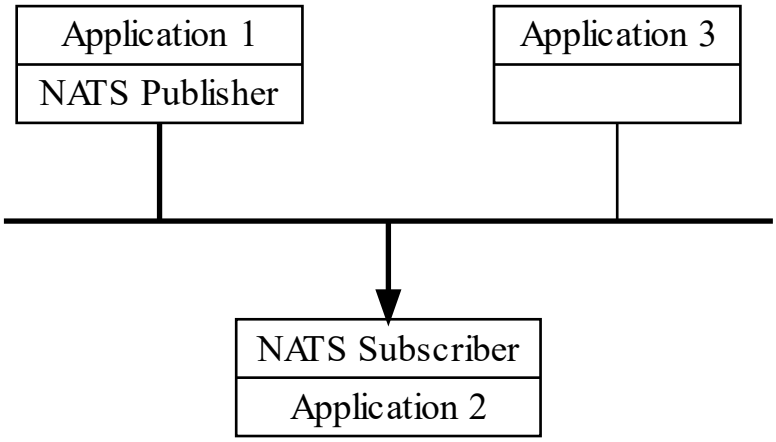
```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 9411
    hostPort: 9411
  - containerPort: 9412
    hostPort: 9412
  - containerPort: 9413
    hostPort: 9413
  - containerPort: 3000
    hostPort: 3000
  - containerPort: 3001
    hostPort: 3001
  - containerPort: 3002
    hostPort: 3002
  - containerPort: 4000
    hostPort: 4000
  - containerPort: 4001
    hostPort: 4001
  - containerPort: 4002
    hostPort: 4002
  - containerPort: 5432
    hostPort: 5432
  - containerPort: 5433
    hostPort: 5433
  - containerPort: 5434
    hostPort: 5434
  - containerPort: 4222
    hostPort: 4222
  - containerPort: 8222
    hostPort: 8222
  - containerPort: 50084
    hostPort: 50084
```

Slika 18: Konfiguracija lokalnog kind klastera

5.2. Komunikacijski sustav *NATS*

Softverske aplikacije i servisi trebaju razmjenjivati podatke. *NATS* je infrastruktura koja [7]omogućuje takvu razmjenu podataka, segmentiranih u obliku poruka. Takav naziv servisa nazivamo *message oriented middleware* [22]. *NATS* nam omogućuje izgradnju raspodijeljenih i skalabilnih aplikacija bez puno napora, i mogućnost pohranjivanja raspodijeljenih podataka u stvarnom vremenu. To se može fleksibilno postići u različitim okruženjima, programskim jezicima, poslužiteljima u oblaku i lokalnim sustavima.

NATS sustav aplikacijama olakšava komunikaciju slanjem i primanjem poruka. Te su poruke adresirane i identificirane nizovima predmeta i ne ovise o mrežnoj lokaciji. Podaci su kodirani i uokvireni kao poruka, a šalje ih izdavač. Na drugoj strani, poruku prima, dekodira i obrađuje jedan ili više pretplatnika.



Slika 19: Jednostavan dizajn razmjene poruka [22]

S ovim jednostavnim dizajnom, *NATS* omogućuje programira dijeljenje zajedničkog koda za upravljanje porukama, izolaciju resursa i međuovisnosti, te skaliranje jednostavnim upravljanjem povećanjem volumena poruka, bilo da se radi o zahtjevima servisa ili tokovi podataka [22].

NATS nudi višestruku kvalitetu servisa, ovisno o tome koristi li aplikacija samo *Core NATS* funkcionalnosti, ili također iskorištava dodatne funkcije koje omogućuje *NATS Jetstream*. *Core NATS* nudi *at most once* kvalitetu servisa [22]. Ako pretplatnik ne sluša na

[7] NATS – sustav za razmjenu poruka

predmet ili nije aktivan kada je poruka poslana, poruka neće biti primljena. Ovo je ista razina jamstva koju pruža *TCP/IP* (eng. *Transmission Control Protocol/Internet Protocol*). *Core NATS* je *fire-and-forget* sustav za slanje poruka [22]. Držat će samo poruke u memoriji i nikada neće pisati poruke izravno na disk. Ako trebamo veću kvalitetu servisa (*at least once* i *exactly once*) ili funkcionalnosti kao što je *persistent streaming*, kontrola protoka bez veze i *key/value* pohrana, možemo koristiti *NATS Jetstream*, koji je ugrađen u *NATS* poslužitelj.

Core NATS osnovni je skup funkcionalnosti i kvaliteta servisa koje nudi infrastruktura *NATS* servisa gdje nijedna od instanci *NATS* poslužitelja nije konfigurirana za omogućavanje *Jetstream* sustava [22]. *Core NATS* funkcionalnosti uključuju implementaciju *publish/subscribe* modela distribucije poruka za komunikaciju jedan naprema više, s adresiranjem na temelju predmeta i mogućnosti stavljanja poruka u red čekanja, i *request-reply* obrazac koji je sagrađen na *publish/subscribe* mehanizmu [22]. *Core* NATS koristi *at most once* kvalitetu servisa.

5.2.1. Konfiguracija *NATS* poslužitelja

Kako bi naši servisi mogli međusobno razmjenjivati poruke moraju uspostaviti vezu na poslužitelja koji pokreće *NATS*. Zbog jednostavnosti, u našem projektu smo odlučili pokretati *NATS* kao instancu *Docker* slike (spremnik). Stoga, naši servisi mogu pristupiti *NATS* poslužitelju preko izložene *port* točke koji je postavljen u našoj *kind* konfiguraciji. Sukladno tome, implementirali smo i poslovni paket koji enkapsulira samo povezivanje na *NATS* poslužitelj sa svrhom apstrahiranja detalja i mogućnosti ponovnog korištenja paketa kroz više servisa koji iskazuju potrebu za istim. Na slici *Slika **20***. prikazan je isječak koda koji implementira sve potrebne funkcionalnosti za povezivanje na *NATS* poslužiteljem koji se pokreće u našem klasteru i metodu za pretplaćivanje na određenu temu i kanal.

```go
// Package nats provides support for connecting to NATS server.
package nats

import (
    "time"

    nats "github.com/nats-io/nats.go"
    stan "github.com/nats-io/stan.go"
)

type Config struct {
    ClusterID string
    ClientID  string
    Host      string
}

type NATS struct {
    AckWait time.Duration
    Client  stan.Conn
}

func Connect(cfg Config) (*NATS, error) {
    nc, err := nats.Connect(cfg.Host)
    if err != nil {
        return nil, err
    }

    sc, err := stan.Connect(cfg.ClusterID, cfg.ClientID, stan.NatsConn(nc))
    if err != nil {
        return nil, err
    }

    aw, _ := time.ParseDuration("60s")

    n := NATS{
        AckWait: aw,
        Client:  sc,
    }

    return &n, nil
}
```

Slika 20: Poslovni paket nats i funkcija za povezivanje na NATS poslužitelja

Kako bi omogućili mehanizam pretplate na određene kanale i teme, implementirali smo *Subscribe* metodu nad našom *NATS* strukturom. Osim što prima ime kanale i teme na koju želimo biti pretplaćeni, očekuje i *callback* funkciju, koja će se pozvati u trenutku kada se okine događaj vezan uz našu pretplatu. Za zadane postavke, odlučili smo da naši servisi prilikom uspostave veze na *NATS* poslužitelja prime sve moguće događaje na koje su pretplaćeni, ali su ih propustili. Uz to, postavljamo ručan način obrade potvrda (eng. *acknowledgement*) poruka. Implementaciju navedenih funkcionalnosti možemo vidjeti na slici *Slika **21***.

```
func (n NATS) Subscribe(subject string, queueGroupName string, cb
stan.MsgHandler) error {
    _, err := n.Client.QueueSubscribe(subject, queueGroupName, cb,
        stan.DeliverAllAvailable(),
        stan.SetManualAckMode(),
        stan.AckWait(n.AckWait),
        stan.DurableName(queueGroupName),
    )

    if err != nil {
        return err
    }

    return nil
}
```

Slika 21: Metoda za pretplaćivanje na određenu temu i kanal

5.2.2. Implementacija *publish* mehanizma

Korištenjem prethodno implementiranog mehanizma omogućili smo servisima jednostavno povezivanje na *NATS* poslužitelj i upravljanje istim. Shodno tome, ne moramo se brinuti o nikakvim detaljima implementacije već samo o podacima koje želimo objavljivati. Na slici *Slika **22***. prikazana je jednostavnost konfiguriranja *NATS* poslužitelja za korištenje unutar pojedinog servisa. Sve što je potrebno je kao parametre u funkciju za uspostavu veze poslati ime *NATS* klastera, ime klijenta, i adresu veze na kojoj se nalazi naš *NATS* poslužitelj.

```
    // NATS Support

    // Create connectivity to the NATS server.
    log.Infow("startup", "status", "initializing NATS support", "host",
cfg.NATS.Host)

    n, err := nats.Connect(nats.Config{
        ClusterID: cfg.NATS.ClusterID,
        ClientID:  cfg.NATS.ClientID,
        Host:      cfg.NATS.Host,
    })

    if err != nil {
        return fmt.Errorf("connecting to NATS server: %w", err)
    }
    defer func() {
        log.Infow("shutdown", "status", "stopping NATS support", "host",
cfg.NATS.Host)
        n.Client.Close()
    }()
```

Slika 22: Stvaranje veze na NATS poslužitelj

Unutar dobivene enkapsulirane jedinice nalazi se klijent pomoću kojega možemo obavljati funkcije objavljivanja poruka na određene teme i kanale. Sama adresa instance proslijeđena je do poslovnog sloja aplikacije, što vidimo na slici *Slika **23***, gdje se takva vrsta funkcionalnosti odvija. Na slici *Slika **24*** prikazano je kako koristimo funkcionalnost objavljivanja poruka unutar aplikacijskih sučelja samih servisa.

```
// Core manages the set of API's for post access.
type Core struct {
    store db.Store
    nats  *nats.NATS
}

// NewCore constructs a core for post api access.
func NewCore(log *zap.SugaredLogger, sqlxDB *sqlx.DB, nats *nats.NATS) Core {
    return Core{
        store: db.NewStore(log, sqlxDB),
        nats:  nats,
    }
}
```

Slika 23: Inicijalizacija nats poslužitelja kod poslovnog sloja

```
// Create inserts a new post into the database.
func (c Core) Create(ctx context.Context, np NewPost, now time.Time) (Post,
error) {
    if err := validate.Check(np); err != nil {
        return Post{}, fmt.Errorf("validating data: %w", err)
    }

    dbP := db.Post{
        ID:          validate.GenerateID(),
        Title:       np.Title,
        Description: np.Description,
        UserID:      np.UserID,
        DateCreated: now,
        DateUpdated: now,
    }

    tran := func(tx sqlx.ExtContext) error {
        if err := c.store.Tran(tx).Create(ctx, dbP); err != nil {
            return fmt.Errorf("create: %w", err)
        }
        return nil
    }

    if err := c.store.WithinTran(ctx, tran); err != nil {
        return Post{}, fmt.Errorf("tran: %w", err)
    }

    var buf bytes.Buffer
    enc := gob.NewEncoder(&buf)

    if err := enc.Encode(&dbP); err != nil {
        return Post{}, fmt.Errorf("encoding: %w", err)
    }

    if err := c.nats.Client.Publish("post-created", buf.Bytes()); err != nil {
        return Post{}, fmt.Errorf("pub: %w", err)
    }

    return toPost(dbP), nil
}
```

Slika 24: Objavljivanje poruke na određenu temu i kanal

5.2.3. Implementacija *subscribe* mehanizma

Osim što servisi mogu objavljivati poruke na određene teme i kanale, isti mogu slušati na događaje i obraditi poruke koje zaprime. Proces spajanja na *NATS* poslužitelja je identičan kao i u prethodnom primjeru, ali postoji dodatan korak konfiguracije koji je potreban kako bi se servisi mogli pretplatiti na određene teme i reagirati u trenutku kada se okine pojedini događaj. Na slici *Slika **25*** prikazana je implementacija strukture koja pruža funkcionalnost pretplate, slušanja na određene događaje, te implementaciju logike koja će se izvršiti prilikom okidanja pojedinog događaja.

Ključne strukture kojima mehanizam pretplate treba imati pristup je inicijaliziran *NATS* klijent sa vezom na poslužitelja, i vezu na bazu podataka. Nakon što primimo poruku prilikom okidanja određenog događaja, u našem slučaju ju je potrebno dekodirati iz polja bitova i transformirati u željeni oblik objave. Zatim ju možemo spremiti u našu bazu podataka i/ili dalje obrađivati poruku.

```go
package comment

import (
    "bytes"
    "context"
    "encoding/gob"

    "github.com/dudakovict/social-network/business/core/comment/db"
    "github.com/dudakovict/social-network/business/sys/nats"
    "github.com/nats-io/stan.go"
    "go.uber.org/zap"
)

type Listener struct {
    log   *zap.SugaredLogger
    nats  *nats.NATS
    store db.Store
}

func (l Listener) PostCreated() error {
    l.nats.Subscribe("post-created", "posts", func(m *stan.Msg) {
        buf := bytes.NewReader(m.Data)
        dec := gob.NewDecoder(buf)

        var dbP db.Post

        err := dec.Decode(&dbP)
        if err != nil {
            l.log.Infof("decoding: %w", err)
        }

        if err := l.store.CreatePost(context.Background(), dbP); err != nil {
            l.log.Infof("create: %w", err)
        }
    })

    return nil
}
```

Slika 25: Implementacija strukture koja pruža mehanizam pretplate

Sama inicijalizacija navedenog mehanizma provodi se u poslovnom sloju unutar strukture koja služi za upravljanje sustavima aplikacijskih sučelja dotičnog servisa. Na slici *Slika **26*** prikazana je navedena inicijalizacija potrošača koji sluša na određene događaje.

```
// Core manages the set of API's for comment access.
type Core struct {
    store db.Store
    nats  *nats.NATS
}

// NewCore constructs a core for comment api access.
func NewCore(log *zap.SugaredLogger, sqlxDB *sqlx.DB, nats *nats.NATS) Core {
    c := Core{
        store: db.NewStore(log, sqlxDB),
        nats:  nats,
    }

    l := Listener{
        log:   log,
        nats:  c.nats,
        store: c.store,
    }

    // Listen on events
    l.PostCreated()
    l.PostUpdated()
    l.PostDeleted()

    return c
}
```

Slika 26: Inicijalizacija strukture koja sluša na definirane događaje

5.3. Unarni *RPC* pozivi između servisa

Kao i mnogi *RPC* sustavi, *gRPC* se temelji na ideji definiranja servisa, određivanjem metoda koje se mogu pozvati s njihovim parametrima i povratnim tipovima [6]. Već smo spomenuli da *gRPC* koristi *protocol buffer* kao jezik za definiranje sučelja za opisivanje sučelja servisa i strukture poruka. Na slici *Slika **27*** prikazana je definicija *.proto* datoteke

za servis koji pruža funkcionalnost slanja elektroničke pošte preko protokola *SMTP* (eng. *Simple Mail Transfer Protocol*).

```
syntax="proto3";

package email;

option go_package = "./email";

service Email {
    rpc Send(EmailRequest) returns (EmailResponse) {}
}

message EmailRequest {
    string email = 1;
}

message EmailResponse {
    string message = 1;
}
```

Slika 27: Definicija metoda i poruka za email servis

Kako bi generirali sve potrebne tipove, sučelja i metode potrebne za inicijalizaciju *gRPC* klijenta i poslužitelja možemo koristiti *protoc*, prevoditelj protokola međuspremnika. Ispod se nalazi instrukcija koja generira datoteke sa navedenim sadržajem.

```
SHELL := /bin/bash

email-proto:
    protoc --go_out=. --go_opt=paths=source_relative \
        --go-grpc_out=. --go-grpc_opt=paths=source_relative \
        business/data/email/email.proto
```

Slika 28: Instrukcija koja prevodi .proto datoteku za email servis

Generirane datoteke sadrže atribute, metode i sučelja koja je potrebno definirati i implementirati kako bi mogli inicijalizirati naš *gRPC* poslužitelj i primati unarne *RPC* zahtjeve za slanje elektroničke pošte. Na slici *Slika **29*** prikazan je paket koji implementira navedene zahtjeve i apstrahira funkcionalnost slanja elektroničke pošte od glavnog

programa. Sučelje zahtijeva implementaciju jedne metode koja šalje elektroničku poštu na adresu poslanu u zahtjevu putem protokola *SMTP*.

```go
// Package email provides support for sending e-mail messages
package email

import (
    "context"
    "fmt"
    "net/smtp"

    "github.com/dudakovict/social-network/business/data/email"
    "go.uber.org/zap"
)

type EmailServer struct {
    log     *zap.SugaredLogger
    address string
    sender  string
    auth    smtp.Auth
    email.UnimplementedEmailServer
}

func NewEmailServer(log *zap.SugaredLogger, address string, sender string,
auth smtp.Auth) EmailServer {
    return EmailServer{
        log:     log,
        address: address,
        sender:  sender,
        auth:    auth,
    }
}

func (es *EmailServer) Send(ctx context.Context, req *email.EmailRequest)
(*email.EmailResponse, error) {
    message := []byte("Subject: Social network\n" + "Welcome to my social
network!")
    err := smtp.SendMail(es.address, es.auth, es.sender, []string{req.Email},
message)
    if err != nil {
        return nil, fmt.Errorf("send: %w", err)
    }

    res := &email.EmailResponse{
        Message: "Successfuly sent an email to " + req.Email,
    }

    return res, nil
}
```

Slika 29: Implementacija sučelja definiranih u .proto datoteci email servisa

Na strani poslužitelja, poslužitelj implementira metode koje je deklarirao servis i pokreće *gRPC* poslužitelj za upravljanje klijentskim pozivima. Infrastruktura *gRPC* programskog okvira dekodira dolazne zahtjeve, izvršava servisne metode i kodira servisne odgovore. Na slici *Slika* ***30*** prikazana je inicijalizacija *gRPC* servera, zajedno sa objektom autentifikacije potrebnim za slanje elektroničke pošte.

```
    // Start gRPC Service

    log.Infow("startup", "status", "gRPC service started", "host",
cfg.GRPC.Address)

    conn, err := net.Listen(cfg.GRPC.Network, cfg.GRPC.Address)

    if err != nil {
        return fmt.Errorf("announcing on the local network: %w", err)
    }

    grpc := grpc.NewServer()

    auth := smtp.PlainAuth("", cfg.SMTP.Username, cfg.SMTP.Password,
cfg.SMTP.Host)
    es := es.NewEmailServer(log, cfg.SMTP.Address, cfg.SMTP.Username, auth)

    email.RegisterEmailServer(grpc, &es)

    if err := grpc.Serve(conn); err != nil {
        log.Errorw("shutdown", "status", "gRPC v1 server closed", "host",
cfg.GRPC.Address, "ERROR", err)
    }
```

Slika 30: Inicijalizacija gRPC servera

Na strani klijenta, klijent ima lokalni objekt poznat kao stub koji implementira iste metode kao i servis. Klijent zatim može pozvati te metode na lokalnom objektu, omatajući parametre za poziv u odgovarajuću vrstu poruke po standardu protokola međuspremnika. Slanje zahtjeva prema poslužitelju i vraćanje poslužiteljevog odgovora u formatu protokola međuspremnika prati *gRPC* [7]. Na slici *Slika* ***31*** prikazano je spajanje na *gRPC* poslužitelja i inicijalizacija klijenta.

```
    // Start GRPC Support

    log.Infow("startup", "status", "initializing gRPC support", "host",
cfg.GRPC.Address)

    conn, err := grpc.Dial(cfg.GRPC.Address, grpc.WithInsecure())
    if err != nil {
        return fmt.Errorf("connecting to gRPC: %w", err)
    }

    defer conn.Close()

    client := email.NewEmailClient(conn)
```

Slika 31: Inicijalizacija gRPC klijenta

Kako bi mogli slati zahtjeve prema našem *gRPC* poslužitelju potreban nam je klijent koji razumije kako pričati s tim poslužiteljem. U našem slučaju, zahtjeve želimo vršiti u trenutku kada se korisnik registrira. Stoga, klijent moramo proslijediti do poslovnog sloja gdje se nalazi skup aplikacijskih sučelja za upravljanje korisnicima, što se vidi na slici *Slika **32***.

```
// Core manages the set of API's for user access.
type Core struct {
    store db.Store
    ec    email.EmailClient
    log   *zap.SugaredLogger
}

// NewCore constructs a core for user api access.
func NewCore(log *zap.SugaredLogger, sqlxDB *sqlx.DB, client
email.EmailClient) Core {
    return Core{
        store: db.NewStore(log, sqlxDB),
        ec:    client,
        log:   log,
    }
}
```

Slika 32: Struktura kojoj se prosljeđuje gRPC klijent

Kao i kod uobičajenih poziva funkcija, unarni *RPC* šalje jedan zahtjev poslužitelju i dobiva jedan odgovor natrag. Nakon što klijent pozove stub metodu, poslužitelj dobiva obavijest da je *RPC* pozvan s metapodacima klijenta za ovaj poziv, nazivom metode i navedenim istekom roka ako je primijenjen [7]. Poslužitelj tada može odmah poslati natrag svoje početne metapodatke koji se moraju poslati prije bilo kakvog odgovora, ili pričekati poruku od zahtjeva klijenta. Ono što se dogodi prvo ovisi o aplikaciji. Nakon što poslužitelj dobije poruku od zahtjeva klijenta, obavlja sve što je potrebno za stvaranje i popunjavanje odgovora. Ako je zahtjev uspješno obrađen, odgovor se vraća klijentu s pojedinostima o statusu i izbornim metapodacima. U slučaju da je status odgovora *OK*, klijent dobiva odgovor, čime se završava poziv na strani klijenta.

U našem slučaju, odlučili smo vršiti zahtjeve prema *gRPC* poslužitelju u trenutku korisnikove registracije. Proces započinje tako da korisnik napravi zahtjev prema našem aplikacijskom sučelju sa validnim podacima za registraciju korisnika. Ako je korisnik uspješno registriran i spremljen u bazu podataka, šaljemo unarni *RPC* zahtjev prema *gRPC* poslužitelju za slanje elektroničke pošte na adresu koju smo poslali u zahtjevu. Prilikom neuspjelog zahtjeva, bilježimo grešku i nastavljamo normalan rad funkcije. Na slici *Slika **33*** prikazan je navedeni tok instrukcija.

```
// Create inserts a new user into the database.
func (c Core) Create(ctx context.Context, nu NewUser, now time.Time) (User,
error) {
    if err := validate.Check(nu); err != nil {
        return User{}, fmt.Errorf("validating data: %w", err)
    }

    hash, err := bcrypt.GenerateFromPassword([]byte(nu.Password),
bcrypt.DefaultCost)
    if err != nil {
        return User{}, fmt.Errorf("generating password hash: %w", err)
    }

    dbUsr := db.User{
        ID:           validate.GenerateID(),
        Name:         nu.Name,
        Email:        nu.Email,
        PasswordHash: hash,
        Roles:        nu.Roles,
        DateCreated:  now,
        DateUpdated:  now,
    }

    if err := c.store.Create(ctx, dbUsr); err != nil {
        return User{}, fmt.Errorf("create: %w", err)
    }

    in := email.EmailRequest{
        Email: dbUsr.Email,
    }

    _, err = c.ec.Send(ctx, &in)
    if err != nil {
        c.log.Info(err)
    }

    return toUser(dbUsr), nil
}
```

Slika 33: Funkcija za slanje unarnog zahtjeva prema gRPC poslužitelju

Zaključak

U ovom diplomskom radu, uspoređene su monolitne i mikroservisne arhitekturne paradigme, te sinkroni i asinkroni komunikacijski obrasci. Slijedno tome, istražena je i prikazana sinkrona i asinkrona komunikacija između servisa u studiji slučaja, te je predstavljena mikroservisna arhitektura na temelju implementiranih servisa.

U usporedbi s monolitnom arhitekturom, otkriveno je da mikroservisi uvode složenost, komunikacijske troškove i novi skup problema koje je potrebno riješiti. Međutim, oni također pružaju organizacijske i tehnološke prednosti. Omogućuju razvojnim timovima veliku razinu nezavisnosti tijekom razvoja novih poslovnih funkcionalnosti i korištenje heterogenih programskih alata i jezika. Izbor monolitne ili mikroservisne arhitekture uvelike ovisi o organizaciji, okruženju i slučaju uporabe.

Utvrđeno je da sinkrona i asinkrona komunikaciju imaju svoju ulogu u sustavu, pa čak i mikroservisne arhitekture koje komuniciraju gotovo samo asinkrono, mogu sadržavati sinkrono aplikacijsko sučelje za izlaganje podataka sustava vanjskim servisima, čak i ako je unutarnje stanje sustava u potpunosti vođeno događajima. Sinkroni komunikacijski obrasci obično su jednostavniji za integraciju i komunikaciju, što ih čini dobrim izborom za korištenje kao eksternog servisa koji izlaže podatke za čitanje. Nasuprot tome, asinkrona komunikacija nudi mnoge prednosti u usporedbi sa sinkronom komunikacijom kao što su elastičnost i tolerancija na greške. Viša razina odvojenosti na razini veze i vremena poželjna je osobina u mikroservisnoj arhitekturi. Nudi višu razinu jamstva za eventualnu obradu događaja, što znači da se može posebno dobro nositi s velikim radnim opterećenjima spremanjem događaja u međuspremnik posrednika dok ih potrošači počinju obrađivati. Složenost koju uvodi asinkrona komunikacija, u obliku složenijih komunikacijskih obrasca, obrada samog posla i postavljanja infrastrukture stvara potencijalno veće troškove razvoja.

Uz sinkrone i asinkrone komunikacijske obrasce, predstavili smo i tehnologije za upravljanje složenim komunikacijskim procesima unutar raspodijeljenog sustava. Vidjeli smo da većina aplikacija može imati koristi od istodobnog korištenja *API Gateway-a* i *service mesh-a*. Dok *API Gateway* može pojednostaviti način na koji aplikacija obrađuje vanjske zahtjeve, *service mesh* igra ključnu ulogu u racionalizaciji interne komunikacije.

API Gateway može proslijediti zahtjev određenom mikroservisu, ali taj mikroservis mora komunicirati s drugim mikroservisom, što je slučaj gdje *service mesh* pomaže. Moguće je koristiti *API Gateway* bez *service mesh-a* i obrnuto. Međutim, to je sve rjeđe. Ako želimo povećati agilnost aplikacije i minimizirati trud koji programeri ulažu u upravljanje komunikacijama, poželjno je koristiti oboje.

Literatura

[1] M. Fowler i J. Lewis, »Microservices,« 25 Ožujak 2014.. [Mrežno]. Available: https://martinfowler.com/articles/microservices.html#SynchronousCallsConsideredHarmful. [Pokušaj pristupa 2022.].

[2] S. Newman, Building Microservices, 1. ur., O'Reilly Media, Inc., 2015..

[3] Microsoft, »Communication in a microservice architecture,« 4 Travanj 2022.. [Mrežno]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture. [Pokušaj pristupa 2022.].

[4] C. Harris, »Microservices vs. monolithic architecture,« 2022.. [Mrežno]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. [Pokušaj pristupa 2022.].

[5] B. Reselman, »Synchronous vs. asynchronous microservices communication patterns,« 10 Ožujak 2021.. [Mrežno]. Available: https://www.theserverside.com/answer/Synchronous-vs-asynchronous-microservices-communication-patterns. [Pokušaj pristupa 2022.].

[6] gRPC, »Introduction to gRPC,« 11 Kolovoz 2021.. [Mrežno]. Available: https://grpc.io/docs/what-is-grpc/introduction/. [Pokušaj pristupa 2022.].

[7] gRPC, »Core concepts, architecture and lifecycle,« 2 Kolovoz 2022.. [Mrežno]. Available: https://grpc.io/docs/what-is-grpc/introduction/. [Pokušaj pristupa 2022.].

[8] Wikipedia, »Representational state transfer,« 2 Rujan 2022.. [Mrežno]. Available: https://en.wikipedia.org/wiki/Representational_state_transfer. [Pokušaj pristupa 2022.].

[9] T. Reeder, »Asynchronous messaging patterns,« 10 Listopad 2019.. [Mrežno]. Available: https://blogs.mulesoft.com/api-integration/patterns/asynchronous-messaging-patterns/. [Pokušaj pristupa 2022.].

[10] C. Tozzi, »Service mesh vs. API gateway: Where, why and how to use them,« 15 Rujan 2021.. [Mrežno]. Available: https://www.techtarget.com/searchapparchitecture/tip/Service-mesh-vs-API-gateway-Where-why-and-how-to-use-them. [Pokušaj pristupa 2022.].

[11] D. Mooter, »Deciphering the Difference Between a Service Mesh and API Gateway,« 21 Srpanj 2020.. [Mrežno]. Available: https://levelup.gitconnected.com/deciphering-the-difference-between-a-service-mesh-and-api-gateway-c57e4abec302. [Pokušaj pristupa 2022.].

[12] A. Gillis i J. Montgomery, »service mesh,« Rujan 2021.. [Mrežno]. Available: https://www.techtarget.com/searchitoperations/definition/service-mesh. [Pokušaj pristupa 2022.].

[13] Microsoft, »Ambassador pattern,« 28 Srpanj 2022.. [Mrežno]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador. [Pokušaj pristupa 2022.].

[14] J. Montgomery, »API gateway,« Ožujak 2021.. [Mrežno]. Available: https://www.techtarget.com/whatis/definition/API-gateway-application-programming-interface-gateway. [Pokušaj pristupa 2022.].

[15] R. Hat, »What does an API gateway do?,« 8 Siječanj 2019.. [Mrežno]. Available: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do. [Pokušaj pristupa 2022.].

[16] Microsoft, »Resilient communications,« 4 Srpanj 2022.. [Mrežno]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/resilient-communications. [Pokušaj pristupa 2022.].

[17] S. Vroonland, »Patterns for making your inter-service communication more resilient,« 5 Listopad 2020.. [Mrežno]. Available: https://medium.com/@svroonland/patterns-for-making-your-inter-service-communication-more-resilient-592ec928296b. [Pokušaj pristupa 2022.].

[18] Microsoft, »Retry pattern,« 28 Srpanj 2022.. [Mrežno]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry. [Pokušaj pristupa 2022].

[19] Microsoft, »Circuit Breaker pattern,« 28 Srpanj 2022.. [Mrežno]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker. [Pokušaj pristupa 2022.].

[20] Microsoft, »Reliability patterns,« 13 Prosinac 2021.. [Mrežno]. Available: https://docs.microsoft.com/en-us/azure/architecture/framework/resiliency/reliability-patterns. [Pokušaj pristupa 2022.].

[21] M. McGranaghan i E. Bendersky, »Go by Example: Rate Limiting,« 6 Lipanj 2022.. [Mrežno]. Available: https://gobyexample.com/rate-limiting. [Pokušaj pristupa 2022.].

[22] NATS, »NATS Docs,« 13 Siječanj 2022.. [Mrežno]. Available: https://docs.nats.io/. [Pokušaj pristupa 2022.].

[23] kind, »kind,« 2022.. [Mrežno]. Available: https://kind.sigs.k8s.io/. [Pokušaj pristupa 2022.].

[24] J. Kanjilal, »3 microservices resiliency patterns for better reliability,« 28 Veljača 2020.. [Mrežno]. Available: https://www.techtarget.com/searchapparchitecture/tip/3-microservices-resiliency-patterns-for-better-reliability. [Pokušaj pristupa 2022.].

[25] J. Kanjilal, »Unravel asynchronous inter-service communication in microservices,« 23 Rujan 2019.. [Mrežno]. Available: https://www.techtarget.com/searchapparchitecture/tip/Unravel-asynchronous-inter-service-communication-in-microservices. [Pokušaj pristupa 2022.].

[26] K. Doyle, »Why you should use a service mesh with microservices,« 17 Studeni 2021.. [Mrežno]. Available: https://www.techtarget.com/searchitoperations/tip/Why-you-should-use-a-service-mesh-with-microservices. [Pokušaj pristupa 2022.].

[27] C. Richardson, »Building Microservices: Using an API Gateway,« 15 Lipanj 2015.. [Mrežno]. Available: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. [Pokušaj pristupa 2022.].

Popis slika

Popis tablica