Komunikacioni paterni u mikroservisnoj arhitekturi

Teodora Damnjanović[1], Bojana Dimić Surla[2]

***Sadržaj* — Rad se fokusira na istraživanje komunikacionih obrazaca u mikroservisnoj arhitekturi, sa posebnim osvrtom na tehnologije koje omogućavaju nesmetanu interakciju među distribuiranim servisima. Mikroservisi, osim izolacije funkcionalnosti, zahtevaju i efikasnu orkestraciju i koordinaciju, kako bi zajedno ispunili složene poslovne procese. U tom smislu, komunikacioni mehanizmi igraju ključnu ulogu u obezbeđivanju koherentnosti celokupnog sistema. Cilj ovog istraživanja je pružiti sveobuhvatan pregled različitih komunikacionih obrazaca u okviru mikroservisne arhitekture. U radu su obrađeni oblici komunikacije: sinhrona putem REST API-ja, asinhrona s RabbitMQ-om kao posrednikom poruka i visokoperformantna komunikacija putem gRPC-a za direktne pozive udaljenih procedura.**

***Ključne reči* — asinhrona komunikacija, Docker, gRPC, mikroservisi, RabbitMQ, softverske arhitekture, sinhrona komunikacija**

I. UVOD

U savremenom dobu brzog tehnološkog napretka, arhitektura softvera je prošla kroz značajne promene, pri čemu je mikroservisna arhitektura postala široko prihvaćen pristup. Ovaj model razvoja softverskih sistema podrazumeva razlaganje monolitnih aplikacija na manje, nezavisno razvijane servise, od kojih svaki ispunjava specifičnu poslovnu funkciju. Takav arhitektonski pristup omogućava veću skalabilnost, fleksibilnost i ubrzan razvoj [1]. Međutim, efikasnost mikroservisne arhitekture u velikoj meri zavisi od pouzdanosti njenih komunikacionih obrazaca.

Kroz 4 poglavlja ovog rada analizirani su različiti komunikacioni obrazci, uključujući njihove teorijske osnove i praktičnu primenu. Rad se fokusira na prednosti i nedostatke svakog obrasca, pružajući korisne smernice inženjerima

[1]Teodora Damnjanović. Autor, Računarski fakultet, Beograd, Srbija (email: te.damnjanovic@gmail.com).

[2]B. Dimić Surla. Autor, Računarski fakultet, Beograd, Srbija; (email: bdimicsurla@raf.rs)

i studentima zainteresovanim za mikroservisnu arhitekturu, sa ciljem unapređenja dizajna i implementacije efikasnih strategija u ovoj oblasti. Korišćena literatura obuhvata najnovija saznanja i pristupe iz oblasti mikroservisne arhitekture i komunikacionih obrazaca.

II. MIKROSERVISNA ARHITEKTURA

Mikroservis je mala aplikacija koja se može razvijati, skalirati i testirati nezavisno. Termin mikroservisne arhitekture se odnosi na stil u razvoju softvera, u kojem je sistem podeljen na veći broj komponenti koji međusobno sarađuju. Svaki mikroservis obuhvata specifičnu poslovnu funkcionalnost, logiku i komunikacioni interfejs. Svaki mikroservis radi na zasebnom procesu ili čak na zasebnoj mašini i najčešće poseduje sopstvenu bazu podataka, što omogućava timovima da rade na izolovanim komponentama, podstičući paralelni razvoj i brže cikluse iteracija [1].

Mikroservisi najčešće komuniciraju preko mreže, putem HTTP protokola. Interfejs svakog servisa mora biti jasno definisan kako bi mogao biti povezan sa drugim komponentama tog sistema, bilo da su to drugi servisi ili druge aplikacije. Unutrašnja implementacija servisa se može menjati sve dok te izmene ne utiču na spoljašnji interfejs. Mikroservisi razdvajaju ponašanje aplikacije na funkcionalne elemente, koji se razvijaju nezavisno i omogućavaju automatizaciju razvoja softvera (eng. continuous integration).

Decentralizacija sistema omogućava nezavisni razvoj i raspoređivanje odnosno agilnost, skalabilnost i autonomiju prilikom razvoja. Istovremeno ona zahtava pažljivo odabran obrazac komunikacije kako bi se obezbedila kohezija celog sistema.

Izazov mikroservisne arhitekture leži u orkestiranju nesmetane komunikacije između delova sistema kako bi se održala celoukupna koherencija sistema[1]. Stoga, efikasnost celog sistema u velikoj meri zavisi od efikasnosti obrazaca za komunikaciju. Sam obim komunikacije između mikroservisa može dovesti do zagušenja mreže, ugrožavajući efikasnost sistema. Takođe, način raspoređivanja i skaliranja mikroservisa utiče na efikasnost komunikacije. Razumevanje ovih izazova je neophodno za osmišljavanje efikasnih komunikacionih obrazaca koji ublažavaju i otklanjaju probleme povezane sa mrežom.

U monolitnoj arhitekturi, komunikacija između različitih delova aplikacije obično se odvija putem direktnih poziva metoda ili funkcija unutar istog procesa, bez korišćenja HTTP/HTTPS protokola. Međutim, kada se monolit razdvaja na mikroservise, HTTP/HTTPS komunikacija postaje ključna za

razmenu podataka između nezavisnih servisa.

U distribuiranom okruženju mikroservisa, standardni HTTP/HTTPS pozivi donose izazove u vezi sa skalabilnošću zbog mrežnih latencija i složenosti upravljanja konekcijama. Da bi se prevazišli ovi izazovi, usvojeni su RESTful API-ji i RPC kao standardizovani mehanizam za komunikaciju [2]. RESTful API omogućava mikroservisima da komuniciraju putem standardizovanih interfejsa bez potrebe za održavanjem sesije, što doprinosi većoj fleksibilnosti i skalabilnosti komunikacione infrastrukture.

Istovremeno, industrija je svedočila pojavi poziva udaljenih procedura (RPC) kao alternativne komunikacione paradigme u mikroservisnoj arhitekturi. RPC je ponudio direktniji i sinhroni pristup komunikaciji. Ovaj pristup pruža neposrednost prilikom poziva metoda, dok je istovremeno uvodio izazove povezane sa potencijalno čvrstom povezanosti između mirkoservisa.

Potreba za prevazilaženjem izazova koje sinhrona komunikacija nosi dovodi do usvajanja redova poruka i uvođenja asinhrone komunikacione paradigme u ekosistem mikroservisa. Sinhrona komunikacija zahteva da servis čeka odgovor pre nego što nastavi sa izvršavanjem, dok asinhrona omogućava da se izvršavanje nastavi odmah nakon slanja zahteva [3]. Redovi poruka omogućili su mikroservisima da komuniciraju bez stroge vremenske povezanosti, rešavajući izazove povezane sa vremenom i redosledom zavisnosti. Ova inovacija najavila je fleksibilniji model komunikacije.

Razumevanje istorijske putanje komunikacije u mikrouslogama je od suštinske važnosti za razumevanje motiva iza savremenih komunikacionih obrazaca i izazova koji ih prate. Pri izboru tehnologije važno je razmotriti koja ograničenja dolaze uz tu tehnologiju, jer je moguće da se način razmišljanja koji stoji iza te tehnologije ne poklapa sa problemom koji pokušavamo da rešimo.

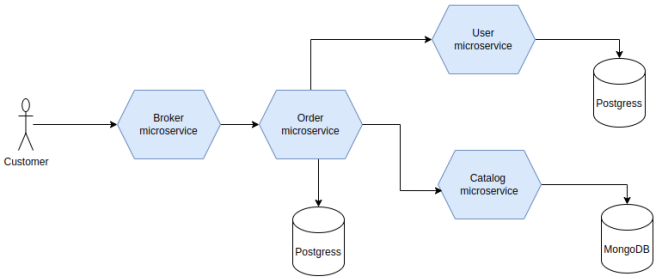
III. SINHRONO BLOKIRAJUĆI ŠABLON KOMUNIKACIJE

Sinhrona komunikacija podrazumeva da servis koji šalje zahtev čeka (ili blokira) dok ne dobije odgovor od servisa kojem je poslao zahtev. Ovaj obrazac se često implementira koristeći HTTP/REST ili gRPC, gde pozivatelj šalje zahtev i čeka odgovor pre nego što nastavi svoje izvršavanje. Blokirajući aspekt znači da je nit koja poziva zaustavljena dok se ne dobije odgovor, što može uvesti latenciju i smanjiti odziv sistema ako se ne upravlja pravilno.

RESTful API-ji, iako široko prihvaćeni, uvode izazove povezane sa njihovom stateless prirodom. Svaki zahtev od jednog mikroservisa ka drugom nosi potrebne informacije za obradu, što dovodi do povećanog mrežnog

saobraćaja. Ova stalna razmena informacija može dovesti do neefikasnosti u scenarijima gde je potrebno više interakcija da bi se ispunila jedna operacija, to se posebno odnosi na spajanje podataka koji su raspoređeni između različitih mikroservisa a potrebno ih je obraditi u okviru jedne operacije.

Posmatrajmo sledeći primer, korisnik želi da pregleda svoju porudžbinu. Tri mikroservisa su uključena u izvršavanje ovog korisničkog zahteva. **Order** mikroservis prima zahtev za dohvatanje porudžbine, on se zatim obraća **Customer** mikroservisu kako bi dohvatio podatke o kupcu, a **Catalog** mikroservisu kako bi se odredila cena za svaki artikal iz porudžbine (Sl. 1.). Prikaz porudžbine ne može da se prikaže sve dok svi zahtevi nisu obrađeni od strane drugih mikroservisa. Kako se zahtevi drugim servisima izvršavaju sinhrono i sekvencijalno, latencije se akumuliraju.



Sl. 1. Sinhrono blokirajući šablon.

Troškovi mrežnih upita, koji se javljaju prilikom pristupa podacima iz drugog mikroservisa, nazivaju se troškovi daljinskog pristupa (eng. Remote Access Costs - RAC). Ovaj problem se, na primer, može rešiti keširanjem. Keširanje se može implementirati između baze i biznis modela unutar svakog mikroservisa u sistemu, čime se smanjuje vreme lokalnog pristupa (eng. Local Access Costs - LAC). Ovakva vrsta keširanja u manjoj meri otklanja problem jer će klijentski servis, **Order** mikroservis, i dalje snositi trošak mrežnog transfera. Da bi se smanjilo vreme mrežnog transvera u mikroservisnoj arhitekturi obično se preporučuje keširanje na nivou međusloja (eng. Middleware caching). Keširanje na nivou međusloja može značajno smanjiti troškove mrežnih upita i poboljšati performanse sistema [4]. Alati za keširanje na nivou međusloja izlažu podatke drugim servisima isključivo za potrebe čitanja, obećavajući visok učinak u brzini čitanja. Takođe, ne skladište relacione podatke, što im omogućava lako skaliranje, potencijalno pružajući brze odgovore iz desetina klastera ako je potrebno. Uprkos ovim nedostacima, sinhrona komunikacija je jednostavna i laka za implementaciju, što je čini

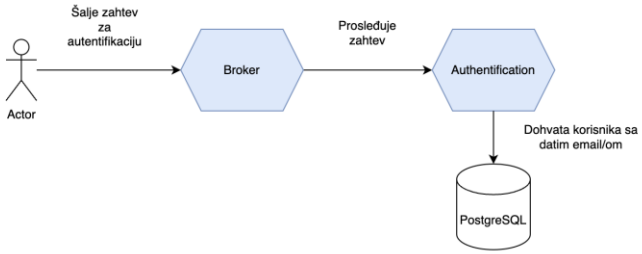
dobrim izborom za mnoge scenarije gde su potrebni odgovori u realnom vremenu.

Prednosti sinhronog blokirajućeg obrasca su **jednostavnost, trenutna povratna informacija i konzistentnost.** Sinhroni pozivi su jednostavni za razumevanje i implementaciju. Model zahtev-odgovor se dobro uklapa u tradicionalne programske paradigme. Trenutna povratna informacija znači da servis koji poziva odmah zna rezultat svog zahteva, što pojednostavljuje rukovanje greškama i upravljanje stanjem. Konzistentnost osigurava da je zahtev obrađen i da je odgovor primljen, smanjujući rizik od problema sa konačnom konzistentnošću koji mogu nastati u asinhronim sistemima.

Izazovi sinhronog blokirajućeg obrasca su **latencija, otpornost na greške i skalabilnost**. Servis koji poziva mora čekati na odgovor, što može povećati latenciju, posebno ako je mreža spora ili je servis koji se poziva zauzet. Otpornost na greške se odnosi na situaciju kada servis koji se poziva ne radi ili ima problema, servis koji ga poziva je direktno pogođen, što može dovesti do kaskadnih grešaka. Usled velikog povećanja sinhronih poziva javlja se potreba za skaliranjem aplikacije, međutim zbog blokirajuće prirode sinhrone komunikacije efikasno skaliranje može biti problem.

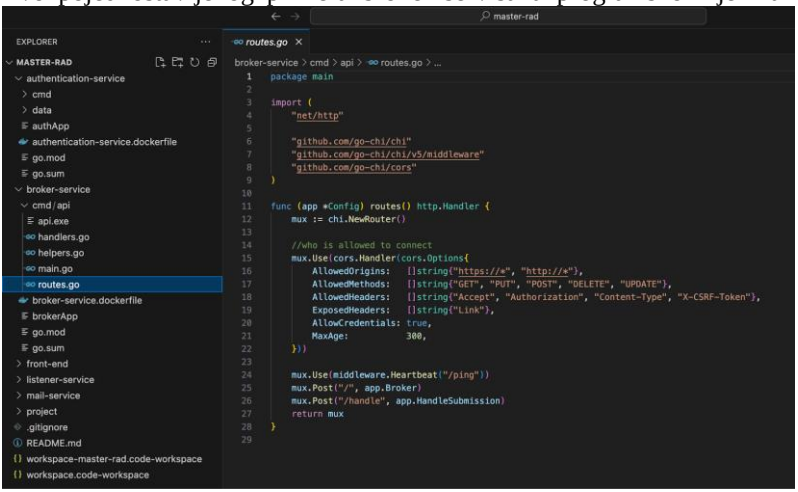
A. Implementacija sinhrone blokirajuće komunikacije u programskom jeziku Go

Da bismo ilustrovali sinhronu blokirajuću komunikaciju, koristićemo dva servisa: Broker servis i servis za Autentifikaciju (auth servis). Broker servis šalje zahteve Authentication servisu da autentifikuje korisnike. Na Sl. 2. prikazan je klijent koji šalje zahtev za autentifikaciju, on je blokiran dok ne dobije odgovarajući odgovor. Ovaj poziv je sinhron blokirajući jer Broker servis čeka odgovor od Authentication servisa kako bi prosledio odgovor ka klijentu. Ovakva vrsta komunikacije može da bude opasna ukoliko dođe do problema sa kašnjenjem mreže ili ukoliko iz nekog razloga Authentication servis sporo reaguje, što može da dovede do ranjivosti celog sistema obzirom da je Broker servis posrednik za komunikaciju u našem sistemu. Zato će u sledećem poglavlju biti uvedena asinhronu komunikaciju između Brokera i ostalih mikroservisa u sistemu.



Sl. 2. Sinhrona komunikacija

Broker servis deluje kao posrednik koji obrađuje dolazne zahteve i prosleđuje ih odgovarajućem servisu, u ovom slučaju Authentication servisu. Evo pojednostavljenog primera broker servisa u programskom jeziku Go.



Sl. 3. Definisanje ruta Broker servisa

Broker servis ima izložen endpoint /handle (Sl. 3.). HandleSubmission funkcija u zavisnosti od akcije koja se nalazi u JSON zahtevu preusmerava dalje zahtev (Sl. 4.).

```go
func (app *Config) HandleSubmission(w http.ResponseWriter, r *http.Request) {
    var requestPayload RequestPayload

    err := app.readJSON(w, r, &requestPayload)
    if err != nil {
        app.errorJSON(w, err)
        return
    }

    switch requestPayload.Action {
    case "auth":
        app.authenticate(w, requestPayload.Auth)
    case "signup":
        app.signup(w, requestPayload.User)
    case "mail":
        app.sendMail(w, requestPayload.Mail)
    default:
        app.errorJSON(w, errors.New("unknown action"))
    }
}
```

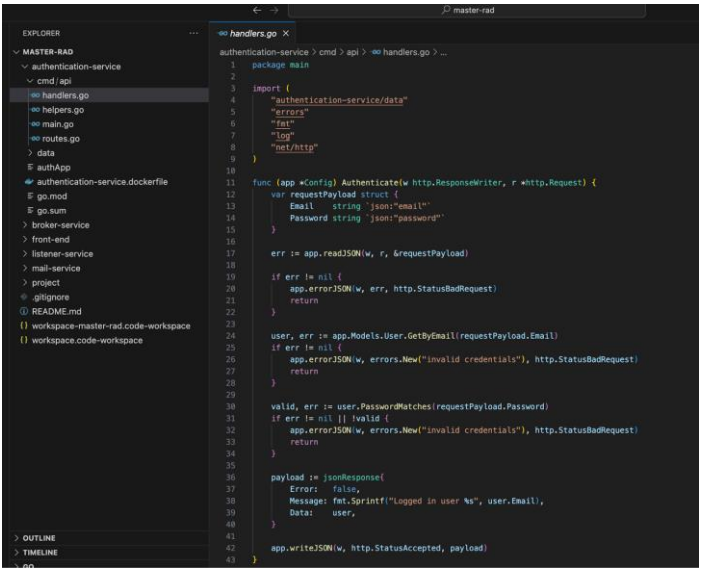
Sl. 4. Obrađivanje poziva Broker servisa

Funkcija **authenticate** šalje POST zahtev authentication servisu sa korisničkim imenom i lozinkom (Sl. 5.).

```go
func (app *Config) authenticate(w http.ResponseWriter, a AuthPayload) {
    jsonData, _ := json.MarshalIndent(a, "", "\t")

    request, err := http.NewRequest("POST", "http://authentication-service/authenticate", bytes.NewBuffer(jsonData))
    if err != nil {
    }

    client := &http.Client{}
    response, err := client.Do(request)
    if err != nil {
    }

    defer response.Body.Close()

    if response.StatusCode == http.StatusUnauthorized {
        app.errorJSON(w, errors.New("invalid credentials"))
        return
    } else if response.StatusCode != http.StatusAccepted {
        app.errorJSON(w, errors.New("error calling auth service"))
        return
    }

    var jsonFromService jsonResponse

    err = json.NewDecoder(response.Body).Decode(&jsonFromService)
    if err != nil {
    }

    if jsonFromService.Error {
        app.errorJSON(w, err)
        return
    }

    var payload jsonResponse
    payload.Error = false
    payload.Message = "Authenticated"
    payload.Data = jsonFromService.Data

    app.writeJSON(w, http.StatusAccepted, payload)
}
```

Sl. 5. Broker servis - Autentifikacija

Funkcija **authenticate**, Authentication servisa, obrađuje dolazne zahteve za autentifikaciju (Sl. 6.). Izvršava jednostavnu proveru i vraća poruku o uspehu ako su kredencijali tačni, u suprotnom vraća grešku.



Sl. 6. Authentication servis - Provera identiteta korisnika

IV. ASINHRONO NEBLOKIRAJUĆI ŠABLON KOMUNIKACIJE

Asinhrona neblokirajuća komunikacija je ključni obrazac u arhitekturi mikroservisa, omogućavajući servise da međusobno komuniciraju bez potrebe za trenutnim odgovorom. Za razliku od sinhrone blokirajuće komunikacije, gde servis koji šalje zahtev mora čekati odgovor pre nego što nastavi sa radom, asinhrona komunikacija omogućava servisu da odmah nastavi sa izvršavanjem drugih zadataka nakon što je zahtev poslat. Ovaj pristup koristi asinhrone mehanizme kao što su poruke, redovi poruka i događaji, što omogućava servisu koji šalje zahtev da bude obavešten o rezultatu operacije kada postane dostupna, bez potrebe za stalnim čekanjem. To rezultira većom skalabilnošću i otpornošću sistema, jer smanjuje vreme čekanja i omogućava bolju iskorišćenost resursa. Iako redovi poruka uvode asinhronu dimenziju u komunikaciju mikrousloga, oni donose i sopstveni skup izazova. Upravljanje redosledom i integritetom poruka postaje ključni aspekt. Osiguravanje da se događaji obrađuju u ispravnom redosledu i da nijedan podatak ne bude izgubljen tokom prenosa zahteva pažljivu koordinaciju. Asinhrona komunikacija, iako korisna, zahteva robusnu infrastrukturu koja će se nositi sa složenostima koje uvodi. U asinhronoj komunikaciji, servisi često koriste posrednike poruka kao što su Apache Kafka ili RabbitMQ za slanje i primanje poruka. Ovaj obrazac je posebno koristan u scenarijima gde je brzina odgovora

nepredvidiva ili gde je potrebno obraditi veliki broj zahteva bez uskih grla.

A. Komunikacija preko zajedničkih podataka

Šablon komunikacija preko zajedničkih podataka se implementira tako što jedan mikroservis upisuje podatke na određenu lokaciju, a drugi mikroservisi zatim koriste te podatke tako što pristupaju datoj lokaciji i sa nje čitaju podatke. Jedan primer ovakve komunikacije ilustrovan je kroz primenu u radnom okruženju na sledeći način:

Prilikom podnošenja aplikacije, Insurance Application Service šalje zahtev Papyrus servisu za kreiranje PDF dokumenta koji predstavlja tu aplikaciju. Papyrus servis zatim komunicira s nekoliko drugih servisa kako bi konstruisao PDF dokument i sačuvao ga na određenoj lokaciji na AWS-u. U međuvremenu, klijentska aplikacija svakih 10 sekundi proverava status kreiranja dokumenta pomoću funkcije **pollForTaskStatus**. Na Sl. 7. prikazan je primer implementacije mehanizma za polling u JavaScript aplikaciji.

```
async pollForTaskStatus(
    id: UUID,
    abortSignal?: AbortSignal,
    maxPollingRetries = 120,
    pollingRetryIntervalInMilliseconds = 1000,
): AsyncResult<
    PollForTaskStatusResponse,
    InvalidArgument | OperationFailed | Aborted | Timeout
> {
    if (maxPollingRetries !== Math.round(maxPollingRetries) || maxPollingRetries < 1) {
        return Failure(
            InvalidArgument({ argument: 'maxPollingRetries', value: maxPollingRetries }),
        );
    }
    let retryCount = 0;
    let isAborted = false;
    if (abortSignal) {
        abortSignal.onabort = () => {
            isAborted = true;
        };
    }
    while (retryCount < maxPollingRetries) {
        if (isAborted) {
            return Failure(Aborted('Polling aborted.'));
        }
        retryCount++;
        const getTaskStatusResult = await this.getTaskStatus(id);
        if (isErr(getTaskStatusResult)) {
            return getTaskStatusResult;
        }
        if (getTaskStatusResult.value) {
            return Success(getTaskStatusResult.value);
        }
        await Async.sleep(pollingRetryIntervalInMilliseconds);
    }
    return Failure(Timeout('Polling max retries exceeded.'));
}
```

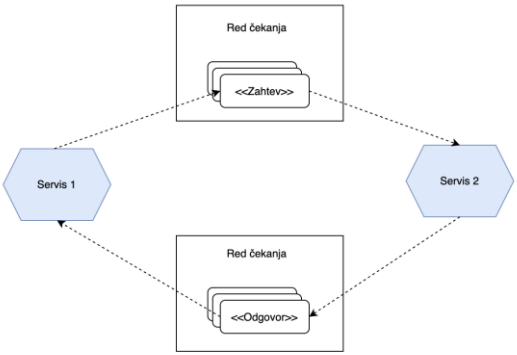
Sl. 7. Provera statusa zadatka

Kada se proces kreiranja PDF-a završi, server ažurira resurs sa konačnim statusom i rezultatom, postavljajući link do dokumenta u zaglavlju ili telu odgovora. Klijent zatim prati taj link kako bi preuzeo kreirani PDF fajl.

B. Komunikacija zahtev odgovor

Asinhroni neblokirajući zahtev odgovor obrazac se koristi u distribuiranim sistemima za upravljanje interakcijama između razlicitih komponenti sistema ili servisa [1]. Ono što razlikuje ovaj obrazac od sinhronog blokirajućeg zahtev odgovor obrasca jeste što kod sinhronog obrasca prilikom slanja zahteva otvara se mrežna veza i zahtev se šalje nizvodnom servisu, prilikom čega veza ostaje otvorena sve dok nizvodni servis ne vrati odgovor. Nizvodni mikroservis ne mora da ima znanje o uzvodnom mikroservisu koji mu je poslao zahtev, jednostavno samo odgovara na zahtev. Problem nastaje kod zahteva koji imaju veliko vreme izvršavanja, tada se mrežna veza prekida i nizvodni mikroservis nema način da pošalje odgovor. Asinhroni neblokirajući obrazac je pogodan način za rešavanje ovakvog problema. Funkcioniše na sledeći način: klijent inicira operaciju slanjem zahteva serveru, koji ga prihvata i kreira resurs koji predstavlja operaciju. Ovaj resurs ima identifikator, status i opciono rezultat ili grešku, a server vraća odgovor ‘’202 Prihvaćeno’’ i vezu ka API endpointu za praćenje statusa. Klijent zatim periodično ispituje tu API krajnju tačku da bi proverio status resursa. Kada se operacija završi i resurs je spreman za preuzimanje, server ažurira resurs sa konačnim statusom i rezultatom ili greškom, postavljajući vezu do njega u zaglavlju ili telu resursa. Klijent prati ovu vezu da bi dobio rezultat ili grešku.

Druga mogućnost da se ovakav problem reši je korišćenje redova čekanja (Sl. 8.). Prilikom ovakve vrste komunikacije servis koji je primio zahtev mora da zna gde da usmeri odgovor, bilo implicitno ili da mu se kaže prilikom slanja zahteva. Prednost korišćenja redova čekanja u odnosu na prethodni pristup jeste što u redovu čekanja može da bude baferovano više zahteva koji čekaju da budu obrađeni. To može da pomogne u situacijama kada zahtevi ne mogu dovoljno brzo da se obrade. Mikroservis može da iskoristi sledeći zahtev kada bude spreman, umesto da bude preopterećen sa previše poziva. Tada dosta zavisi od implementacije reda čekanja. Međutim, kada mikroservis dobije zahtev na ovaj način, može biti neophodno da poveže odgovor sa originalnim zahtevom. Ovo može predstavljati izazov jer je možda prošlo mnogo vremena, a u zavisnosti od upotrebljenog protokola, može se desiti da odgovor ne stigne istoj instanci mikroservisa koja je poslala zahtev. Jednostavno rešenje bi bilo sačuvati stanje povezano sa originalnim zahtevom u bazu podataka, tako da instanca koja primi odgovor može da učita povezano stanje i reaguje na odgovarajući način.



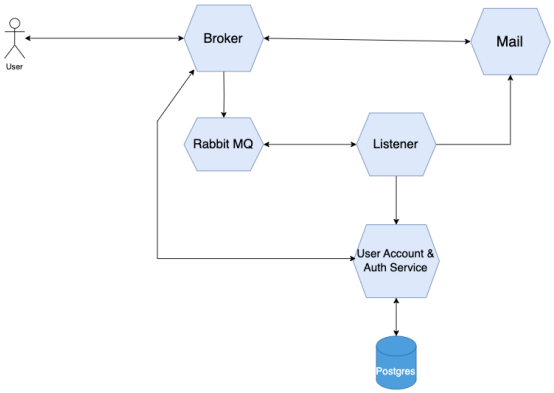
Sl. 8. Šablon asinhrono neblokirajuće komunikacije [1]

Posebno je važno obratiti pažnju na optimizaciju i upravljanje greškama. Kada je potrebno izvršiti više poziva različitim mikroservisima kako bi se nastavila dalja obrada, poželjno je paralelno izvršavati te pozive. Na taj način, ukupno vreme čekanja zavisi od najsporijeg API-ja, umesto da bude zbir vremena svih poziva. Takođe, neophodno je implementirati neku vrstu upravljanja pauzama (eng. timeout) kako bi se izbegli problemi kada sistem ostaje blokiran, čekajući nešto što se nikad neće desiti, odnosno čekajući na odgovor koji neće stići.

C. Interakcija vođena događajem

Interakcija vođena događajem omogućava mikroservisima da šalju događaje bez znanja o servisima koji ih slušaju, čime se smanjuje njihova povezanost [4]. Ključna razlika između ovog i prethodnog obrasca je u tome da servis koji šalje događaj nema svest o servisima koji će taj događaj da slušaju i o akcijama koje će nakon toga da se izvrše.

Posmatrajmo opet primer autentifikacije korisnika (Sl. 9.). Kada klijent pošalje zahtev za autentifikaciju, broker taj zahtev šalje na RabbitMQ broker poruka. Listener servis osluškuje poruke i prosleđuje ih ka odgovarajućem servisu. Authentication servis autentifikuje korisnika i šalje događaj na RabbitMQ obaveštavajući da se pošalje mail, nakon čega Listener servis prosleđuje tu poruku ka Mail servisu.



Sl. 9. Interakcija vođena događajem

Servis za autentifikaciju koristi RabbitMQ da pošalje mejl nakon kreiranja korisničkog naloga, pozivom metode **sendMailEventViaRabbit** (Sl. 10)

```
type MailMessage struct {
    From    string `json:"from"`
    To      string `json:"to"`
    Subject string `json:"subject"`
    Message string `json:"message"`
}
func (app *Config) sendMailEventViaRabbit(user data.User) {
    msg := MailMessage{
        From:    "master@rad.com",
        To:      user.Email,
        Subject: "User registration service",
        Message: "Hi, " + user.FirstName + " you are successfuly signed up!",
    }
    err := app.pushToQueue(msg)
    if err != nil {... }
}
type Payload struct {
    Name string      `json:"name"`
    Data MailMessage `json:"data"`
}
func (app *Config) pushToQueue(msg MailMessage) error {
    emitter, err := event.NewEventEmitter(app.Rabbit)
    if err != nil {... }
    payload := Payload{
        Name: "send-mail",
        Data: msg,
    }
    j, _ := json.MarshalIndent(&payload, "", "\t")
    err = emitter.Push(string(j), "mail.send")
    if err != nil {... }
    return nil
}
```

Sl. 10. Slanje poruke na RabbitMQ u servisu za autentifikaciju

Struktura **MailMessage** sadrži osnovne informacije o pošiljaocu, primaocu, naslovu i sadržaju poruke. Funkcija **sendMailEventViaRabbit** kreira poruku dobrodošlice za novog korisnika i šalje je pomoću metode **pushToQueue**. Ova metoda koristi NewEventEmitter za slanje poruke na **mail.send** kanal putem RabbitMQ. Struktura **Payload** omogućava formatiranje podataka pre slanja, gde **Name** definiše tip poruke, a **Data** sadrži kompletan **MailMessage**. Ova implementacija omogućava slanje notifikacija putem RabbitMQ u realnom vremenu nakon registracije korisnika.

```
package main
import (...)
func (app *Config) Authenticate(w http.ResponseWriter, r *http.Request) {
   var requestPayload struct {
       Email    string `json:"email"`
       Password string `json:"password"`
   }
   err := app.readJSON(w, r, &requestPayload)
   if err != nil {
       app.errorJSON(w, err, http.StatusBadRequest)
       return
   }
   user, err := app.Models.User.GetByEmail(requestPayload.Email)
   if err != nil {
       app.errorJSON(w, errors.New("invalid credentials"), http.StatusBadRequest)
       return
   }
   valid, err := user.PasswordMatches(requestPayload.Password)
   if err != nil || !valid {
       app.errorJSON(w, errors.New("invalid credentials"), http.StatusBadRequest)
       return
   }
   payload := jsonResponse{
       Error:   false,
       Message: fmt.Sprintf("Logged in user %s", user.Email),
       Data:    user,
   }
   app.writeJSON(w, http.StatusAccepted, payload)
}
```

Sl. 11.Autentifikacija korisnika u servisu za autentifikaciju

Na Sl. 11. prikazana je funkcija **Authenticate** iz servisa za autentifikaciju u Go jeziku. Ova funkcija prihvata HTTP zahtev s korisničkim podacima, proverava da li korisnik postoji u bazi i da li su kredencijali ispravni. Ako je autentifikacija uspešna, funkcija vraća odgovor sa podacima o korisniku; u suprotnom, vraća grešku. Ovaj deo koda predstavlja ključni deo procesa autentifikacije u servisu, dok ostale komponente pružaju podršku za komunikaciju sa RabbitMQ i slanje događaja

```
func (app *Config) Signup(w http.ResponseWriter, r *http.Request) {
   var requestPayload struct {
       Email     string `json:"email"`
       FirstName string `json:"first_name,omitempty"`
       LastName  string `json:"last_name,omitempty"`
       Password  string `json:"password"`
   }
   err := app.readJSON(w, r, &requestPayload)
   if err != nil {...   }
   user := data.User{
       Email:     requestPayload.Email,
       FirstName: requestPayload.FirstName,
       LastName:  requestPayload.LastName,
       Password:  requestPayload.Password,
       Active:    1,
   }
   userId, err := app.Models.User.Insert(user)
   if err != nil {... }
   app.sendMailEventViaRabbit(user)
   payload := jsonResponse{
       Error:   false,
       Message: fmt.Sprintf("%s is signed up", user.Email),
       Data:    userId,
   }
   app.writeJSON(w, http.StatusAccepted, payload)}
```

Sl. 12. Prijavljivanje korisnika u servisu za autentifikaciju

Na Sl. 12. prikazana je funkcija **Signup** u servisu za autentifikaciju, koja upravlja procesom registracije novog korisnika. Funkcija prima HTTP zahtev s podacima korisnika (Email, FirstName, LastName, i Password), koje zatim validira i koristi za kreiranje novog korisničkog zapisa u bazi. Nakon uspešnog unosa, funkcija poziva **sendMailEventViaRabbit** za slanje obaveštenja putem RabbitMQ, a kao odgovor vraća JSON poruku s informacijama o uspešnoj registraciji korisnika.

```
import (...)
type Emitter struct {
   connection *amqp.Connection
}
func (e *Emitter) setup() error {
   channel, err := e.connection.Channel()
   if err != nil { return err }
   defer channel.Close()
   return declareExchange(channel)
}
func NewEventEmitter(conn *amqp.Connection) (Emitter, error) {
   emitter := Emitter{
       connection: conn,
   }
   err := emitter.setup()
   if err != nil { return Emitter{}, err }
   return emitter, nil
}
```

Sl. 13. Inicijalizacija EventEmitter-a i deklaracija razmene za RabbitMQ

Na Sl. 13. prikazan je deo koda servisa za autentifikaciju u kojem se definiše

struktura **Emitter**, koja upravlja povezivanjem s RabbitMQ brokerom poruka. Funkcija **NewEventEmitter** deluje kao konstruktor za Emitter, uspostavljajući vezu putem *amqp.Connection. Metoda **setup** otvara kanal za komunikaciju i poziva funkciju **declareExchange**, koja konfiguriše razmenu (exchange) s parametrima za ime, tip (topic), i trajnost (durable). Ako setup ili declareExchange naiđu na grešku, vraćaju je; inače, uspešna konekcija omogućava Emitter-u da šalje poruke na mail_topic razmenu, prikazanu na Sl. 14.

```
func declareExchange(ch *amqp.Channel) error {
   return ch.ExchangeDeclare(
       "mail_topic", // name
       "topic",      // type
       true,         // durable?
       false,        // auto-deleted?
       false,        // internal?
       false,        // no-wait?
       nil,          // arguements?
   )}
```

Sl. 14. Deklaracija razmene (exchange) za RabbitMQ

Na Sl. 15. prikazana je funkcija `Push` iz **Emitter** strukture koja šalje poruku na RabbitMQ. Funkcija prima parametre **event** i **severity** (označava prioritet ili tip poruke) i uspostavlja kanal na postojećoj konekciji. Nakon uspešnog otvaranja kanala, kreira kontekst s vremenskim ograničenjem od 5 sekundi za slanje poruke. Poruka se zatim objavljuje na **mail_topic** exchange-u s prosleđenim parametrima. U slučaju greške, funkcija beleži grešku i vraća je; ako je uspešna, vraća **nil**, što označava uspešno slanje.

```
func (e *Emitter) Push(event string, severity string) error {
   channel, err := e.connection.Channel()
   if err != nil {
       return err
   }
   defer channel.Close()
   ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
   defer cancel()
   log.Println("Pushing to channel")
   err = channel.PublishWithContext(
       ctx,
       "mail_topic",
       severity,
       false,
       false,
       amqp.Publishing{
           ContentType: "text/plain",
           Body:        []byte(event),
       },
   )
   if err != nil {
       log.Println("Push to chanel err ", err)
       return err
   }
   return nil}
```

Sl. 15. Slanje poruke putem funkcije Push u EventEmitter-u

Na Sl. 16a prikazana je definicija strukture **Consumer** i inicijalizacija veze sa RabbitMQ. Struktura Consumer sadrži polja za povezivanje (conn) i naziv reda (queueName). Funkcija **NewConsumer** kreira instancu Consumer i poziva **setup** metodu, koja otvara komunikacioni kanal sa RabbitMQ. U slučaju greške prilikom otvaranja kanala ili deklaracije razmene, funkcija vraća grešku, čime se osigurava stabilna konekcija za dalje primanje poruka.

```
type Consumer struct {
    conn      *amqp.Connection
    queueName string
}
type MailMessage struct {
    From    string `json:"from"`
    To      string `json:"to"`
    Subject string `json:"subject"`
    Message string `json:"message"`
}
func NewConsumer(conn *amqp.Connection) (Consumer, error) {
    consumer := Consumer{conn: conn}
    err := consumer.setup()
    if err != nil { return Consumer{}, err }
    return consumer, nil
}
func (consumer *Consumer) setup() error {
    channel, err := consumer.conn.Channel()
    if err != nil {
        return err
    }
    return declareExchange(channel)
}
```

Sl. 16. Konzumiranje i obrada poruka u Listener servisu

Na Sl. 17 prikazana je funkcija **Listen** koja omogućava Consumer-u da kontinuirano prati i konzumira poruke iz RabbitMQ. Funkcija otvara kanal i kreira red za navedene teme (topics), koji su povezani sa **mail_topic** razmenom. Svaka poruka je asinkrono obrađena pozivom **handlePayload**, što omogućava efikasnu obradu poruka u realnom vremenu.

```go
type Payload struct {
    Name string      `json:"name"`
    Data MailMessage `json:"data"`
}
func (consumer *Consumer) Listen(topics []string) error {
    ch, err := consumer.conn.Channel()
    if err != nil { return err }
    defer ch.Close()
    q, err := declareRandomQueue(ch)
    if err != nil { return err }
    for _, s := range topics {
        ch.QueueBind(
            q.Name,
            s,
            "mail_topic",
            false,
            nil,
        )
        if err != nil { return err }
    }
    messages, err := ch.Consume(q.Name, "", true, false, false, false, nil)
    if err != nil { return err }
    forever := make(chan bool)
    go func() {
        for d := range messages {
            var payload Payload
            _ = json.Unmarshal(d.Body, &payload)

            go handlePayload(payload)
        }
    }()
    fmt.Printf("Waiting for message [Exchange, Queue] [mail_topic, %s]\n", q.Name)
    <-forever
    return nil
}
```

Sl. 17. Funkcija Listen za praćenje i konzumiranje poruka

Na Sl. 18. prikazana je funkcija **handlePayload**, koja obrađuje poruke u zavisnosti od vrednosti **payload.Name**. Ako je naziv poruke **auth**, poziva se authEvent, dok **send-mail** pokreće sendMailEvent za slanje mejla. Funkcija sendMailEvent sastavlja i šalje HTTP zahtev ka mailer-service, konvertujući podatke iz poruke u JSON format. U slučaju greške, funkcija beleži grešku i vraća je; uspešan odgovor potvrđuje uspešno poslatu poruku.

```go
func handlePayload(payload Payload) {
    switch payload.Name {
    case "auth":
        err := authEvent(payload)
        if err != nil {
            log.Println("Consumer_AuthError", err)
            return
        }
    case "send-mail":
        err := sendMailEvent(payload)
        if err != nil {
            log.Println("Consumer_SendMailError", err)
            return
        }
    default:
    }
}
func sendMailEvent(entry Payload) error {
    jsonData, _ := json.MarshalIndent(entry.Data, "", "\t")
    mailServiceURL := "http://mailer-service/send"
    request, err := http.NewRequest("POST", mailServiceURL, bytes.NewBuffer(jsonData))
    if err != nil {
        return err
    }
    request.Header.Set("Content-Type", "application/json")
    client := &http.Client{}
    response, err := client.Do(request)
    if err != nil {
        return err
    }
    defer response.Body.Close()
    if response.StatusCode != http.StatusAccepted {
        return err
    }
    return nil
}
```

Sl. 18. Listener servis: Obrada poruka

U Docker compose fajlu se nalazi konfiguracija RabbitMQ servisa koji je izložen na portu 5672 (Sl. 19.).

```yaml
 image: 'rabbitmq:3.11.13-alpine'
    ports:
      - "5672:5672"
    deploy:
      mode: replicated
      replicas: 1
    volumes:
      - ./db-data/rabbitmq/:/var/lib/rabbitmq/
```

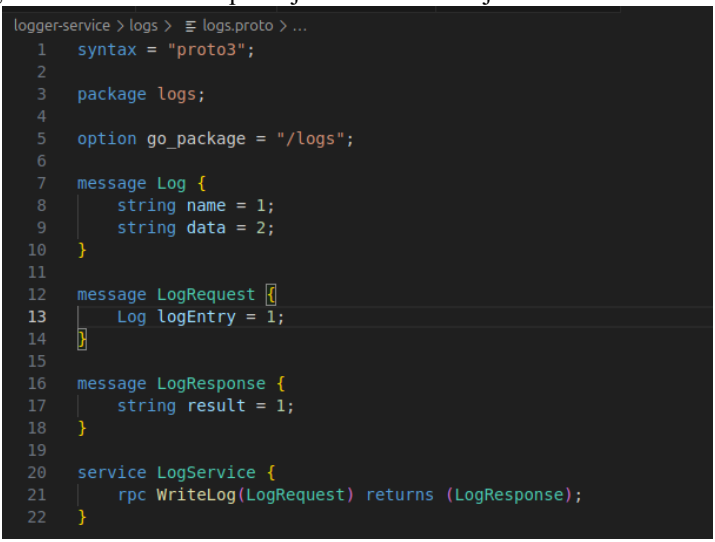
Sl. 19. Konfiguracija RabbitMQ servisa u Docker Compose-u

V. PROTOKOL GRPC U MIKROSERVISNOJ ARHITEKTURI

Protokol gRPC je visokoperformantni, otvoreni RPC okvir koji omogućava brzu komunikaciju između servisa napisanih u različitim programskim jezicima. Za razliku od REST API-ja, gRPC koristi HTTP/2 i Protocol Buffers

za binarnu serijalizaciju, što povećava brzinu prenosa podataka i smanjuje režijske troškove. Jedna od ključnih prednosti gRPC-a je njegova podrška za različite oblike strimovanja, kao što su jednosmerno, dvosmerno, klijentsko i serversko strimovanje.

Kod gRPC API-ja, klijent može slati više zahteva i primati odgovore paralelno, što ga čini pogodnim za aplikacije u realnom vremenu i visokoperformantne sisteme. Server definiše strukturu podataka u Protocol Buffer (proto) fajlovima, koji se zatim koriste za generisanje klijentskog i serverskog koda u različitim programskim jezicima (Sl. 20). Ovaj pristup omogućava konzistentnu tipizaciju i ubrzava razvoj.



Sl. 20. Definisanje logs.proto fajla

Prednosti gRPC-a u mikroservisima uključuju visoke performanse zahvaljujući binarnoj serijalizaciji i korišćenju HTTP/2 protokola, što smanjuje latenciju i povećava efikasnost komunikacije. Takođe, gRPC nudi poliglotsku podršku jer omogućava razvoj na više programskih jezika, što olakšava rad u heterogenim okruženjima. Uz to, gRPC pruža ugrađenu podršku za dvosmerno strimovanje, što ga čini idealnim za zahtevne aplikacije koje se oslanjaju na strimovanje podataka u realnom vremenu.

U poređenju s RPC-om, gde se zahtev brže obrađuje kada su podaci jednostavniji, gRPC pokazuje prednost kod složenijih podataka zahvaljujući optimizovanom protokolu. Rezultati testova pokazuju da gRPC može biti manje efikasan u scenarijima s malim podacima, ali značajno korisniji kod velikih datasetova ili kodova sa složenim interakcijama.

VI. ZAKLJUČAK

Arhitektura mikroservisa, po svojoj prirodi, podložna je kontinuiranoj evoluciji i prilagođavanju. Kako se pojavljuju novi zahtevi i sistemi skaliraju, zahtevi za komunikacionim obrascima prolaze kroz dinamične promene. Suština problema leži u pronalaženju delikatne ravnoteže između agilnosti, skalabilnosti i održavanja kohezivne komunikacije među mnoštvom mirkoservisa. Evolucija mikroservisa podstiče stalnu potrebu za istraživanjem komunikacionih metodologija koje mogu pratiti promenljivo okruženje. Tradicionalni obrasci mogu imati poteškoća u prilagođavanju evoluirajućim potrebama složenih ekosistema mikroservisa, što zahteva kritičku evaluaciju savremenih rešenja.

LITERATURA

[1] *S. Newman, Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015*.
[2] *E. Wolff, Microservices: Flexible Software Architecture, O'Reilly Media, Inc., 2016*.
[3] M. Richards, Fundamentals of Software Architecture: An Engineering Approach, O'Reilly Media, Inc., 2020
[4] *Nginx Documentation, https://docs.nginx.com*.
[5] M. Fowler, Event-Driven Architecture, martinfowler.com, 2017.

ABSTRACT

The paper focuses on the exploration of communication patterns in microservice architecture, with special emphasis on technologies that enable seamless interaction among distributed services. In addition to isolating functionalities, microservices require efficient orchestration and coordination in order to collectively fulfill complex business processes. In this sense, communication mechanisms play a key role in ensuring the coherence of the entire system. The aim of this research is to provide a comprehensive overview of different communication patterns within the context of microservice architecture. This paper covers synchronous communication through simple REST APIs, followed by asynchronous communication using RabbitMQ as a message broker.

Communication patterns in microservice architecture

Teodora Damnjanović, Bojana Dimić Surla