Omejevanje frekvence zahtevkov na odjemalcu

Aljaž Mislovič

Podatkovni inženiring & zaledni sistemi, Databox, Ptuj
aljaz.mislovic@outlook.com

Sinopsis Strežniki so pogosto tarča namernega ali nenamernega povišanja frekvence zahtevkov v kratkem časovnem obdobju. Z različnimi metodami omejevanja zato strežniki ščitijo svoje sistemske vire, podatkovne baze in izboljšujejo uporabniško izkušnjo. V članku smo reševali realni problem produkcijskega okolja Databox, ki v vlogi odjemalca z različnih strežnikov črpa podatke za svoje uporabnike. V prispevku smo opisali, kako smo zasnovali in implementirali programsko rešitev, ki uspešno omejuje frekvenco zahtevkov na razne strežnike z različnimi metodami omejevanja. Na koncu smo predstavili tudi rezultate testiranja na več sto milijonov izvedenih zahtevkov na produkcijskem okolju, ter podali rezultate zmogljivosti.

Ključne besede:
frekvenca zahtevkov
omejevanje zahtevkov
odjemalec
drsno okno
API

Opomba: Prispevek temelji na: Mislovič, A. (2022). Sistem za omejevanje frekvence zahtevkov na odjemalcu : diplomsko delo, Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor: A. Mislovič.

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-639-6
DOI https://doi.org/10.18690/um.feri.10.2022.9

1 Uvod

Najbrž smo vsi kdaj stopili v dvigalo in opazili opozorilni napis za maksimalno število ljudi oziroma maksimalno skupno težo. Te omejitve proizvajalci nastavijo, da zaščitijo dvigalo in ljudi v njem. Spletne storitve uporabljajo podobno omejitev, ki jo imenujemo *omejevanje frekvence zahtevkov*. S tem zaščitijo sebe in svoje uporabnike.

Z napredkom tehnologije in povezovanjem različnih sistemov na spletu hitro narašča tudi količina prenesenih podatkov. Velik delež teh podatkov se prenaša s pomočjo spletnih strežnikov. Spletne storitve množični dostop do podatkov zagotavljajo preko aplikacijskega programskega vmesnika (angl. Application Programming Interface, API). Povečana količina poizvedb pa zahteva tudi določeno mero zaščite pred različnimi namernimi ali nenamernimi zlorabami kot je npr. prevelika frekvenca zahtev. Spletne storitve se pred slednjo zaščitijo s pomočjo sistemov za omejevanje frekvence zahtevkov, ki so lahko implementirani z uporabo različnih algoritmov. Odjemalci, ki pošiljajo veliko količino zahtevkov na isti strežnik, morajo tako sami poskrbeti za omejevanje.

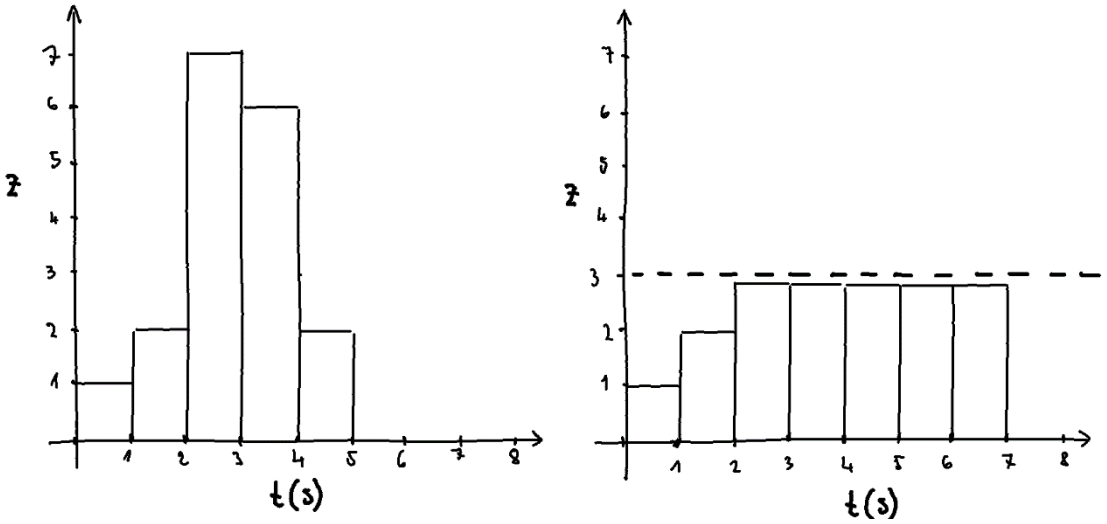
V tem članku, ki temelji na diplomskem delu [5], primarno opisujemo reševanje realnega problema na produkcijskem okolju podjetja Databox [Databox, glej diploma]. Databox podpira pridobivanje podatkov iz več kot 70-ih spletnih storitev, več kot polovica le-teh pa implementira eno ali več različnih metod omejevanja zahtevkov. Na spletne strežnike z omejitvami dnevno izvede okrog 17 milijonov zahtevkov. Razporejanje teh zahtevkov znotraj omejitev je zahtevno opravilo z vidika sistemskih virov, zato smo želeli implementirati sistem, ki bo kos nalogi.

V nadaljevanju članka smo podrobneje predstavili kontekst problema. Teoretično smo razdelali nekaj najpogostejših metod omejevanja frekvence zahtevkov, s katerimi se redno srečujemo. Opisanim metodam smo nato poiskali skupni imenovalec, ter pripravili načrt za implementacijo. S psevdokodo smo implementacijo tudi predstavili. Ob koncu pa smo podali še nekaj rezultatov s produkcijskega okolja.

2 Omejevanje zahtevkov na strani odjemalca

Omejevanje zahtevkov se običajno implementira na strani ponudnika storitve in predstavlja učinkoviti način za preprečevanje širjenja napak v porazdeljenih sistemih. Ena izmed primerov uporabe omejevanja zahtevkov je zaščita pred napadi onemogočanja storitve (angl. Denial of Service), s katerim nepridipravi preplavijo nek strežnik z zahtevki. Podobno lahko tudi odjemalci nenamerno preplavijo sistem. To se lahko zgodi zaradi napak v kodi ali pa s kodo, ki istočasno pošilja ogromne količine zahtevkov. Drug primer uporabe so spletne storitve za prenos reprezentativnega stanja (angl. representational state transfer, REST), ki uporabljajo podatkovno bazo, za zaščito le-te pa razvijalci storitve implementirajo omejevanje zahtevkov. Brez omejevanja bi lahko storitev, ki je zmožna sprejeti več zahtevkov hkrati, povzročila ogromno količino sočasnih zahtevkov, podatkovne baze pa niso opremljene z jasnim načinom omejevanja frekvence zahtevkov. Vse to lahko upočasni spletno storitev, poveča odzivni čas ali pa spletno storitev naredi začasno nedostopno [1].

Spletne storitve, ki omejevanja zahtevkov nimajo implementiranega, torej nimajo kontrole nad uporabo le-te s strani odjemalcev, s tem odjemalcem dovolijo prosto uporabo in so dovzetna za zgoraj opisane situacije. Pomembnost omejevanja zahtevkov smo ponazorili s spodnjo sliko. Slika 1 na levi prikazuje primer števila zahtevkov brez omejevanja, na desni pa pa enako količino zahtevkov, ki se zaradi omejitve treh zahtevkov na sekundo razporedijo skozi čas.



Slika 1: Primerjava zahtevkov na spletno storitev brez in z omejevanjem zahtevkov.
Vir: lasten.

Drugi, manj tipičen način omejevanja je omejevanje zahtevkov na strani odjemalca, ki pa je v našem primeru potreben. Več kot 70 različnih podatkovnih virov se na različne načine integrira v Databox platformo. Vsaka od integracij podpira številne metrike in omogoča pretok veliko podatkov, kar se preslika v ogromno količino zahtevkov, ki jih izvedemo.

Omejevanje zahtevkov je mogoče izvajati čisto reaktivno ali proaktivno. Primer reaktivnega omejevanja zahtevkov je odziv na informacijo klicane spletne storitve, da zahtevke pošiljamo s preveliko frekvenco - torej po tem, ko se zgodi napaka. V tem primeru moramo velikokrat počakati, da mine določen čas, ki je specificiran s strani ponudnika kot omejitev, zato da ne pride do prevelike obremenitve storitve. V veliko primerih ta potreben čas čakanja ni podan v povratni informaciji ob napaki. V našem primeru smo se osredotočili predvsem na drugi, proaktivni način, s katerim poskrbimo, da strežnikov prekomerno ne obremenjujemo in da do napake sploh ne pride.

V Databoxu integracije implementirajo integracijski inženirji, ki morajo pred dejansko implementacijo vsako integracijo podrobno analizirati, raziskati, ter najdene metode omejevanja tudi opisati v primernem formatu, da ga lahko strojno obdelujemo in uporabimo. Najpogosteje najdene metode smo predstavili v nadaljevanju.

3 Metode omejevanja zahtevkov

Kot smo omenili že prej, je namen omejevanja zahtevkov med drugim tudi zaščita same spletne storitve ter njenih sistemskih virov. Najpogosteje razvijalci to storijo z implementacijo ene ali več metod omejevanja frekvence zahtevkov, opisanih v nadaljevanju. Preden pa se razvijalci omejevanja frekvence zahtevkov na spletnih storitvah lotijo izbire prave metode, morajo razumeti, zakaj omejitve implementirajo. Ključnega pomena je, da razumejo svoj sistem in se vprašajo, po katerem ključu (angl. limiting key) bodo implementirali omejitve. Namreč to informacijo morajo razumeti tudi odjemalci.

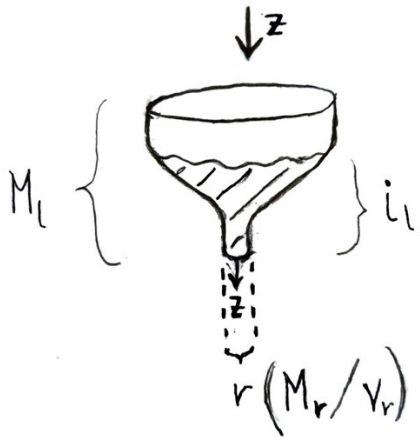
Strežniške sistemske vire si v večini primerov deli več odjemalcev oziroma uporabnikov. Povišano število zahtevkov enega odjemalca lahko nenamerno (ali namerno) vpliva na ostale odjemalce. V ta namen je izbira ključa omejevanja pomembna za zagotavljanje enakomerne in poštene delitve sistemskih virov. Omejitve oziroma kvote razvijalci nastavijo na število zahtevkov v nekem časovnem obdobju. Pri nekaterih strežnikih so rezervirane ali porabljene kvote lahko tudi plačljive [1].

Primeri ključev so identifikator (angl. identifier, ID) odjemalca (angl. client ID), ID uporabnika (angl. user ID), naslov spletnega protokola (angl. internet protocol address, IP address), ID vira, za katerega zahtevamo podatke, itd. Ko je ključ izbran, lahko začnemo slediti uporabi servisa za vsak ključ in izberemo metodo za omejevanje frekvence zahtevkov [1]. V nadaljevanju smo opisali tri najpogostejše metode, s katerimi se integracijski inženirji na Databoxu najpogosteje srečujejo.

3.1 Puščajoče vedro

Puščajoče vedro (angl. *Leaky Bucket*) je metoda, ki jo lahko opišemo z analogijo vedra ali lijaka. Lijak prepušča neko maksimalno količino vode v določenem času. Prav tako ima lijak maksimalen volumen vode, ki je lahko istočasno v lijaku. Na zgornji strani v lijak prilivamo vodo. Če vodo prilivamo hitreje, kot jo lijak prepušča, bo lijak preplavljen.

V primeru strežniških zahtevkov lahko algoritem puščajočega vedra implementiramo kot vrsto FIFO (angl. first in, first out). Odjemalec zahtevke iz vrste procesira z ritmom r, ki ga definiramo s številom zahtevkov M_r v časovni enoti v_r. Vrsta ali lijak lahko naenkrat hrani neko maksimalno število zahtevkov M_l, prav tako pa hrani števec, označen z i_l. Števec nam pove trenutno količino zahtevkov, shranjenih v vrsti. Zahtevek, ki ga želimo shraniti v vrsto, smo označili s črko z. Ponazoritev lahko vidimo na sliki 2.



Slika 2: Ponazoritev metode »puščajoče vedro«.
Vir: lasten.

Sliko 2 smo ponazorili še s spodnjo psevdokodo.

Psevdokoda 1: Omejevanje zahtevkov z metodo »puščajoče vedro«

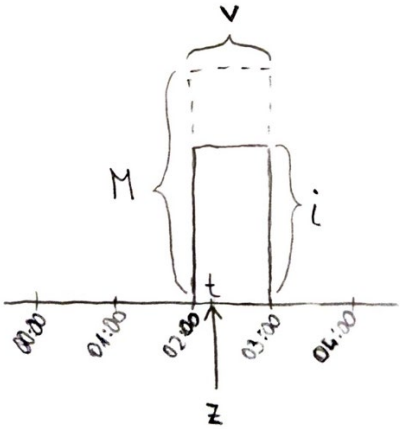
```
če i + z <= M
    zahtevek shrani v vrsto
sicer
    zahtevek zavrzi ali počakaj, da nekaj zahtevkov iz vrste zaključi s
procesiranjem
```

Prednost te metode je, da se zahtevki iz vrste pošiljajo v procesiranje s približno isto hitrostjo, ne glede na to, če zahtevke shranjujemo s konstantno ali povišano hitrostjo. Vendar pa ima ta metoda tudi pomanjkljivost, saj ne zagotavlja, da se procesiranje zahtevka zaključi v nekem konstantnem času. Shranjevanje povišanega števila zahtevkov v vrsto v kratkem času pomeni, da bo vsak naslednji zahtevek na procesiranje čakal dlje, saj je v vrsti ostalo nekaj starejših zahtevkov [2].

3.2 Fiksno okno

Metoda za omejevanje zahtevkov fiksno okno (angl. fixed window) uporablja fiksno velikost časovnega okna oziroma časovni interval, v katerem definiramo maksimalno število zahtevkov. Za lažje razumevanje lahko tudi to metodo opišemo s terminologijo iz prejšnje metode, terminologijo vedra. Pri metodi fiksnega okna vedro ne pušča, ampak se sprazni ob določenem času – na primer, vodo ves dan zlivamo v vedro, ki ga spraznimo vsako jutro ob točno določenem času. Če v nekem dnevu v vedro zlijemo več vode, kot je njegov volumen, bo voda tekla čez rob [1].

Strežniki za implementacijo te metode najprej definirajo velikost okna ***v***. Velikosti so najpogosteje nastavljene v človeku berljivi obliki, kot so na primer 1 minuta, 1 ura ali 1 dan. Prav tako definirajo maksimalno število zahtevkov ***M***, ki jih lahko v tem oknu izvedemo na strežnik. Tudi pri tej metodi je implementiran števec ***i***, ki nam pove število zahtevkov, ki jih je uporabnik izvedel v določenem oknu. Zahtevek, ki ga odjemalec pošlje ob nekem času ***t***, smo označili s črko ***z***. Metodo na časovnici prikazuje slika 3.



Slika 3: Ponazoritev metode fiksno okno.
Vir: lasten.

Odločitveni model za sprejemanje ali zavračanje zahtevkov se glasi:

Psevdokoda 2: Omejevanje zahtevkov z metodo fiksno okno.

```
če i + z <= M(t)
    zahtevek sprejmi
sicer
    zahtevek zavrzi in počakaj na naslednje časovno okno
```

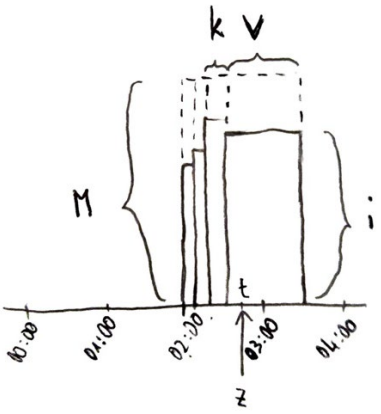
Prednost te metode je enostavna implementacija, ki zahteva zelo malo sistemskih virov, vendar pa nas ta metoda še vedno ne obvaruje pred povišanim številom zahtevkov v zelo kratkem času, ki lahko preobremenijo sistem. Dnevno fiksno okno odjemalcem namreč dovoli, da lahko celotno dovoljeno število zahtevkov izvedejo že v prvi minuti dneva [1].

3.3 Drsno okno

Namen metode drsno okno (angl. sliding window) je čim bolj zgladiti število zahtevkov na strežnik skozi čas. Podobno kot pri metodi fiksno okno tudi drsno okno uporablja velikost časovnega okna oziroma interval, v katerem definiramo maksimalno število zahtevkov. Za razliko od fiksnega okna pa se drsno okno ne »izprazni« le na koncu časovnega okna, temveč se skozi čas pomika oziroma drsi z določenim korakom.

Konec časovnega okna je torej vedno ta trenutek, zaokrožen navzgor na neko časovno enoto, ki jo definira velikost koraka (milisekunde, sekunde, minute, itd.).

Konfiguracija vsebuje – enako kot pri fiksnem oknu – velikost časovnega okna ***v***, le da je z namenom glajenja zahtevkov skozi čas večinoma podana v manjših enotah (1 s, 10 s, 60 s, 100 s ..). Časovno okno skozi čas drsi s korakom ***k***. V časovnem oknu je prav tako definirano maksimalno število zahtevkov ***M***, ki jih strežnik dovoljuje izvesti v časovnem oknu, števec ***i*** pa shranjuje število zahtevkov, ki jih odjemalec naredi v posameznem časovnem oknu. Zahtevek ***z***, ki ga odjemalec naredi ob času ***t***, torej pade v več oken istočasno (odvisno od velikosti časovnega okna in velikosti koraka), kar prikazuje spodnja slika.



Slika 4: Ponazoritev metode drsno okno.
Vir: lasten.

Omejevanje s to metodo smo opisali s psevdokodo 3.

Psevdokoda 3: Omejevanja zahtevkov z metodo drsno okno.

```
če i + z <= M(t)
    zahtevek sprejmi
sicer
    zahtevek zavrzi in počakaj na zdrs časovnega okna, dokler zahtevki niso pod
omejitvijo
```

Implementacija te metode je nekoliko zahtevnejša, saj je treba časovno okno zamikati in s tem ustrezno posodabljati tudi števec zahtevkov. Časovno okno z drsenjem odjemalce spodbuja k enakomernejšemu pošiljanju zahtevkov na strežnik.

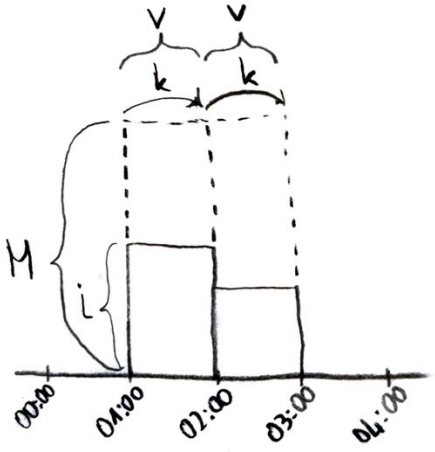
3.4 Skupni imenovalec opisanim metodam

Med raziskavo metod za omejevanje frekvence zahtevkov je postalo jasno, da imajo vse izmed njih nekaj skupnih točk. Med načrtovanjem rešitve smo se začeli spraševati, ali lahko vse tri naštete metode podpremo z eno samo rešitvijo, zato smo jih poskušali dati na skupni imenovalec. Ugotovili smo, da je vsem skupno to, da vsaka izmed njih dovoljuje določeno število zahtevkov v nekem časovnem oknu.

3.4.1 Primerjava metod drsno okno in fiksno okno

Metodi imata za posamezno okno teoretično enako opisano velikost okna ***v***, enako opisano maksimalno kapaciteto okna ***M*** ter števec trenutnega števila zahtevkov ***i***. Razlika je le v drsnem koraku ***k***. Korak nam pove, za koliko enot na časovnici okno drsi. Pri razlagi metode fiksnega okna koraka sicer nismo omenili,

povedali pa smo, da se naslednje okno začne na koncu prejšnjega okna (npr. naslednjo uro ali naslednji dan). Na osnovi tega lahko trdimo, da je korak k pri metodi fiksnega okna enak velikosti okna v. To je možno razbrati tudi s slike 5.



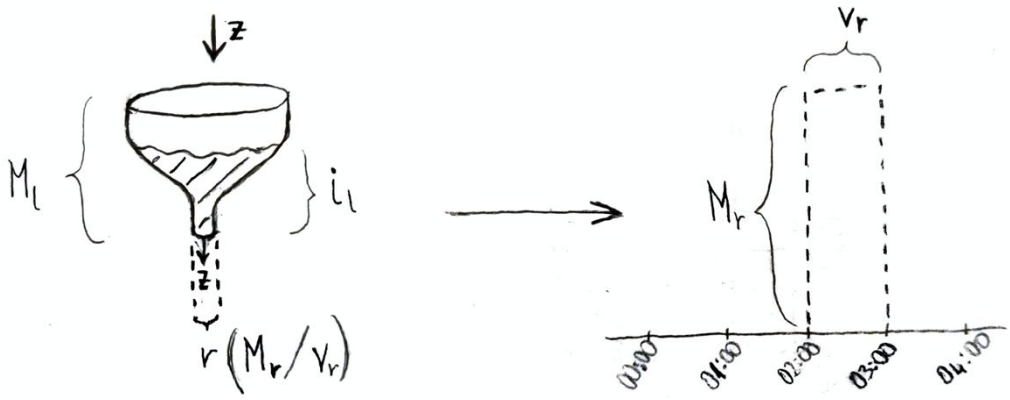
Slika 5: Primerjava metod fiksno okno in drsno okno.
Vir: lasten.

Tako smo ugotovili, da lahko obe metodi damo na skupni imenovalec, če na strani odjemalca implementiramo metodo drsno okno.

3.4.2 Primerjava metod drsno okno in puščajoče vedro

Metoda puščajoče vedro se od metode drsno okno razlikuje predvsem po tem, da ima puščajoče vedro vrsto. Kot smo navedli v opisu metode, je namen te vrste sprejeti večje število zahtevkov v kratkem času, brez vpliva na aplikacijski del strežnika. Procesiranje zahtevkov iz te vrste je določeno z ritmom r. Pomislili smo na primer, ko na strežnik pošiljamo maksimalno število zahtevkov M_r v enaki časovni enoti (oknu) v_r, kot določa ritem r. To pomeni, da vsi zahtevki takoj zapustijo vrsto, saj ne čakajo na nobene predhodne zahtevke. Če ostanemo znotraj teh omejitev, potem vrste oziroma vedra ali lijaka ne potrebujemo.

Naloga naše programske rešitve je, da zna zahtevke pošiljati znotraj nekih omejitev. Če metodi puščajoče vedro v implementaciji odstranimo vrsto, lahko metodo opišemo zgolj z ritmom. Kot smo omenili, je ritem definiran z nekim številom zahtevkov z v neki časovni enoti, na primer 10 z/1 s, 1200 z/60 s, 5000 z/3600 s. Na sliki 6 smo ritem r preslikali na časovnico.



Slika 6: Preslikava metode puščajoče vedro na časovnico.
Vir: lasten.

Ritem r metode puščajoče vedro nas hitro spomni na ponazoritev metode fiksno okno, ki smo jo ponazorili na sliki 3. V prejšnjem podpoglavju smo primerjali metodi drsno okno in fiksno okno ter ugotovili, da je metodo fiksno okno mogoče opisati z metodo drsno okno, menimo, da je tudi metodo puščajoče vedro možno opisati z metodo drsno okno. Z drugačnim oziroma bolj omejenim opisom metode puščajoče vedro smo torej to metodo dali na skupni imenovalec z metodo drsno okno. S tem smo dosegli podporo večim različnim metodam omejevanja z eno samo programsko rešitvijo. V nadaljevanju smo torej načrtovali le podporo metodi drsno okno.

4 Načrtovanje programske rešitve

V prejšnjem poglavju smo navedli, da je možno tri najpogosteje uporabljene metode omejevanja frekvence zahtevkov opisati z metodo drsno okno. Metod za omejevanje obstaja več, kot smo jih našteli. Ena izmed možnosti je bila raziskati vse metode in poiskati skupni imenovalec vseh. To bi povečalo kompleksnost rešitve, podpora vsem pa bi najverjetneje vpeljala dodatno kompleksnost ter dodatne omejitve. V izogib temu smo se odločili, da naša rešitev implementira zgolj metodo drsno okno. Vseh prej omenjenih 70+ integracij, ki jih Databox podpira, so integracijski inženirji uspeli opisati s to metodo.

Pred implementacijo smo iz preteklih izkušenj in implementacij definirali še nekaj drugih predpostavk in omejitev:

- k števcu zahtevkov vedno prištejemo en zahtevek naenkrat (to nam omogoča natančnejše omejevanje);
- predvidevamo, da strežnik in odjemalec števce zahtevkov prištevata v trenutku, ko odjemalec zahtevek izvede (zanemarimo omrežno latenco oz. čas, ki ga zahtevek potrebuje, da doseže strežnik);
- časovno okno v mora biti naravno število;
- časovno okno v je podano v sekundah;
- maksimalno število zahtevkov M mora biti naravno število;
- zgornje meje velikosti časovnega okna ne bomo nastavljali, ker ne poznamo vseh primerov uporabe; dodali jo bomo po potrebi.

4.1 Konfiguracija omejitve frekvence zahtevkov

Naša rešitev mora za delovanje razumeti, s kakšnimi omejitvami dela. Glede na te omejitve smo kasneje implementirali posamezne števce ter integracijskim inženirjem zagotovili, da danih omejitev ne prekoračijo. V konfiguracijo omejitve smo zajeli dva tipa spremenljivk - osnovne in identifikacijske spremenljivke. Osnovne so tiste, ki jih sistem potrebuje za omejevanje.

Ponovno moramo poudariti, da sistem istočasno omogoča uporabo več različnih omejitev in za vsako omejitev več različnih števcev. Opisne spremenljivke tako potrebujemo, da znamo za posamezno integracijo in posameznega uporabnika voditi pravi števec, kjer se nahaja trenutno stanje omejitve.

Končni nabor zahtevanih vhodnih parametrov v sistem za omejevanje je sledeč:

- *id* – spremenljivka, ki služi kot identifikacija posameznega števca omejitve;
- *interval* – spremenljivka, v kateri navedemo velikost časovnega okna v v sekundah;
- *requests* (slov. zahtevki) – spremenljivka, v kateri navedemo maksimalno število dovoljenih zahtevkov v tem časovnem oknu M.

Metoda drsno okno pa za delovanje potrebuje še korak k. Med raziskavo spletnih storitev smo ugotovili, da dokumentacije velikosti koraka ne podajo. Nekatere ne podajo niti algoritma, ki se skriva za omejitvijo. Uporabniku smo želeli poenostaviti uporabo naše rešitve, zato smo se odločili, da bo korak k izračunala koda sama.

4.2 Izračun velikosti koraka

Korak je ključna spremenljivka, ki bo definirala natančnost naše programske rešitve. Pri definiranju le-tega smo se seveda vprašali, kakšna je smiselna in zadovoljiva natančnost. Velikost koraka prav tako določa število zdrsov, ki ga posamezno časovno okno naredi. Več zdrsov pomeni tudi zahtevnejše procesiranje.

Omejitve, ki jih uporabljajo naše integracije, uporabljajo velik nabor intervalov. Najmanjši interval, ki ga med integracijami najdemo, je 1 sekunda, največji pa 1 dan. Iz tega razloga fiksne velikosti koraka za vse omejitve ni bilo smiselno nastaviti. Po preračunavanju smo prišli do zaključka, da bomo velikost koraka računali približno sorazmerno z velikostjo intervala. Pravila za izračun velikosti koraka glede na časovno okno oziroma interval smo zapisali v tabeli 1.

Tabela 1: Velikost koraka glede na velikost intervala.

Interval (od, do]	Korak	Najmanj zdrsov	Največ zdrsov
0 s – 10 s	1 cs	100	1000
10 s – 60 s (1 min)	1 ds	110	1000
60 s (1 min) – 3600 s (1 h)	1 s	61	3600
3600 s (1 h) – 86400 s (1 dan)	1 min	61	1440
86400 s (1 dan) – ∞	1 h	25	∞

4.3 Drsni števci zahtevkov

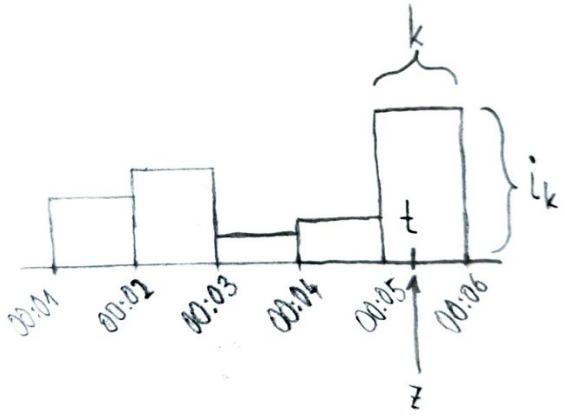
Pri načrtovanju smo izhajali iz razlage za metodo drsno okno v podpoglavju 3.3, kjer smo navedli, da časovno okno v, prej opisano s spremenljivko interval, drsi skozi čas s korakom k. Vsa časovna okna so zaokrožena na velikost koraka, kar definira tudi njihovo natančnost. V nadaljevanju smo posamezno časovno okno opisali z matematično notacijo odprtega in zaprtega intervala – »[začetek okna, konec okna)«. Časovno okno se začne z zaprtim intervalom, kar pomeni, da čas, ki opisuje začetek okna, še zajema

zahtevke, izvedene v tistem trenutku. Konec časovnega okna pa smo opisali z odprtim intervalom. To pomeni, da časovno okno vključuje zahtevke, izvedene vse do trenutka, ki opisuje konec časovnega okna. Zahtevki, izvedeni točno v času konca okna, pa v časovno okno več ne spadajo.

Pri načrtovanju smo našli več načinov implementacije, a na koncu izbrali slednjega: **vodimo števce korakov k_i ter iz njih izračunamo stanje omejitve na zahtevo**. S to rešitvijo ob vsakem zahtevku izvedemo zgolj eno prištevanje. Trenutno stanje omejitve dobimo tako, da seštejemo vse števce korakov, ki spadajo v trenuten interval. Podrobnejši opis in načrt te rešitve smo razdelili na dva dela, to sta prištetje posameznega števca koraka in izračun trenutnega stanja časovnega okna oziroma trenutnega stanja omejitve.

4.4 Prištetje števca koraka

Števec koraka nam pove, koliko zahtevkov je bilo narejenih v časovnem oknu, ki je enako velikosti koraka. Iz časa zahtevka t in velikosti koraka k bo naša rešitev sama določila, kateri števec koraka i_k mora povečati za 1. Na časovnici smo to ponazorili z naslednjo sliko.



Slika 7: Prištetje števca koraka, v katerem je bil zahtevek izveden.
Vir: lasten.

S spodnjo psevdokodo 4 smo opisali celoten postopek prištetja števca koraka.

Psevdokoda 4: Postopek prištetja števca koraka

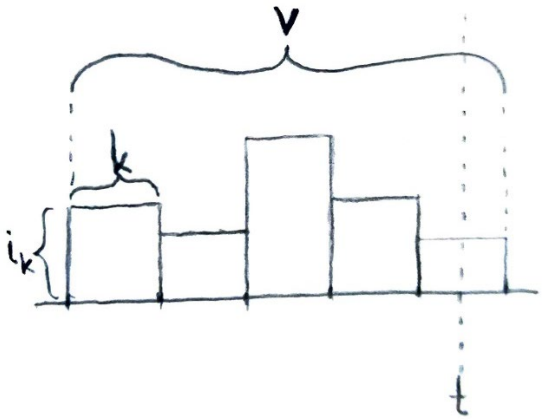
```
t = časZahtevka

v = pridobiIntervalOmejitve() // glej poglavje 4.1
k = izračunajVelikostKoraka(v) // glej poglavje 4.2

idŠtevcaKoraka = izračunajIdPripadajočegaŠtevecaKoraka(k, t)
če !številecKorakaŽeObstaja(idŠtevcaKoraka)
    inicializirajŠtevecKoraka(idŠtevcaKoraka)

povečajŠtevecKoraka(idŠtevcaKoraka)
```

S tem smo dosegli, da naša rešitev zna voditi števce korakov. Uporabnik mora zgolj poklicati metodo vedno, ko izvede zahtevek, in kot parameter podati čas zahtevka. Naslednji del naše rešitve pa je ključen za

omejevanje frekvence zahtevkov. V naslednjem podpoglavju smo opisali, kako naša rešitev izračuna trenutno stanje omejitve.

4.5 Izračun trenutnega stanja omejitve

Trenutno stanje omejitve smo v prejšnjih poglavjih opisali s črko ***i***, ki pomeni trenutno število zahtevkov, narejenih v intervalu omejitve oz. v časovnem oknu ***v***. Ta informacija kasneje Databox platformi pove, če lahko v tem trenutku izvede še kak zahtevek, ali je omejitev že dosežena in moramo še počakati. Za izračun stanja zahtevkov bomo iz konfiguracije omejitve potrebovali velikost časovnega okna ***v***. Prav tako bomo ponovno uporabili prej opisane metode za izračun velikosti koraka ter pripadajočega okna zahtevka. Slika 8 prikazuje števce korakov i_k, ki spadajo v neko časovno okno ***v***, s črko ***t*** pa smo označili trenuten čas, ob katerem izračunavamo stanje omejitve.



Slika 8: Števci koraka v časovnem oknu oziroma intervalu.
Vir: lasten.

Po tej formuli moramo za izračun števca zahtevkov ***i*** sešteti vse števce korakov i_k, ki so element časovnega okna ***v*** v trenutku ***t***. To smo zapisali tudi s psevdokodo 5.

Psevdokoda 5: Izračun trenutnega stanje omejitve.

```
t  = trenutenČas
v = pridobiIntervalOmejitve() // glej poglavje 4.1
vZačetek = izračunajZačetekIntervalaGledeNaČas(v, t)
vKonec = izračunajKonecIntervalaGledeNaČas(v, t)
števciKoraka = pridobiŠtevceKorakaMed(vZačetek, vKonec)
i = seštejŠtevceKoraka(števciKoraka)
```

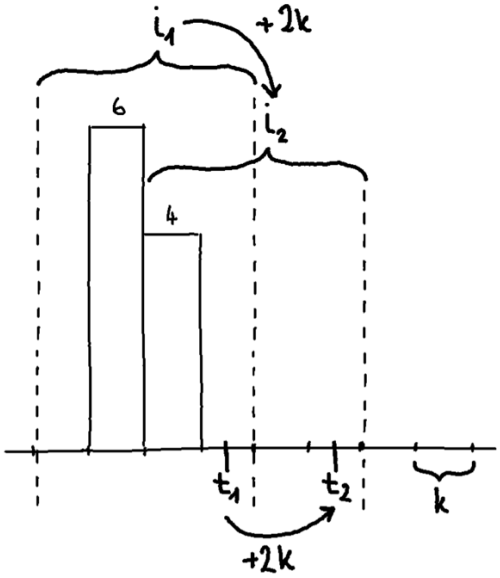
S tem naši rešitvi omogočimo pridobivanje stanja omejitve za trenutni čas. Ko ta seštevek primerjamo z maksimalnim številom zahtevkov ***M***, lahko ugotovimo, kako je stanje posamezne omejitve.

4.6 Izračun naslednjega prostega časovnega okna za izvedbo zahtevka

Z informacijo stanja omejitve se lahko uporabnik v nekem trenutku odloči, če bo izvedel nek zahtevek. Če je odgovor na to vprašanje negativen, mora uporabnik to vprašanje ponavljati, dokler mu izračun stanja ne pove, da lahko zahtevek izvede. Zato smo se odločili našo programsko rešitev izboljšati z odgovorom na vprašanje: »Kdaj lahko izvedem naslednji zahtevek?« V podpoglavju 4.5 smo specificirali metodo, ki zna izračunati trenutno stanje zahtevkov, kot parameter pa prejme čas t. Če ugotovimo, da trenutno stanje

zahtevkov i v časovnem oknu v za čas t ne omogoča izvedbe naslednjega zahtevka, lahko čas t zamaknemo za korak k v prihodnost. Če ta čas podamo v metodo, se začetek časovnega okna na časovnici prestavi za en korak naprej. To efektivno pomeni, da prvi števec koraka odstranimo od prej izračunanega seštevka. Ker števci koraka v prihodnosti ne obstajajo oziroma je njihovo stanje 0, pa ničesar ne dodamo.

Na sliki 9 smo izračun naslednjega prostega časovnega okna za izvedbo zahtevka prikazali na primeru omejitve **10 zahtevkov / 4 časovne enote**. Stanje števca i_1 za trenuten čas t_1 je enako 10, kar pomeni, da je omejitev dosežena. Novih zahtevkov v času t_1 ne smemo izvajati. Ugotovili smo, da moramo t_1 zamaknit za 2 koraka k v prihodnost (t_2), saj bo takrat stanje števca i_2 enako 4, kar pomeni, da lahko izvedemo nove zahtevke.



Slika 9: Zamik okna v prihodnost in iskanje naslednjega okna, v katerem bo možna izvedba zahtevka.
Vir: lasten.

Prej omenjeni postopek zamika smo opisali še s psevdokodo spodaj.

Psevdokoda 6: Izračun naslednjega prostega časovnega okna za izvedbo zahtevka.

```
t  = trenutenČas
v = pridobiIntervalOmejitve() // glej poglavje 4.1
M = pridobiŠteviloDovoljenihZahtevkovOmejitve() // glej poglavje 4.1
k = izračunajVelikostKoraka(v) // glej poglavje 4.2
i = izdačunajStanjeOmejitveVČasu(t) // glej poglavje 4.5
dokler i >= M
    t = t + k
    i = izdačunajStanjeOmejitveVČasu(t) // glej poglavje 4.5
časNaslednjegaOknaZaIzvedboZahtevka = t
```

S tem diagramom poteka smo zaključili načrtovanje naše programske rešitve. Na Databoxu smo to rešitev implementirali kot ločen strežnik, ki hrani informacije o vseh števcih. Strežnik ponuja dve končni točki.

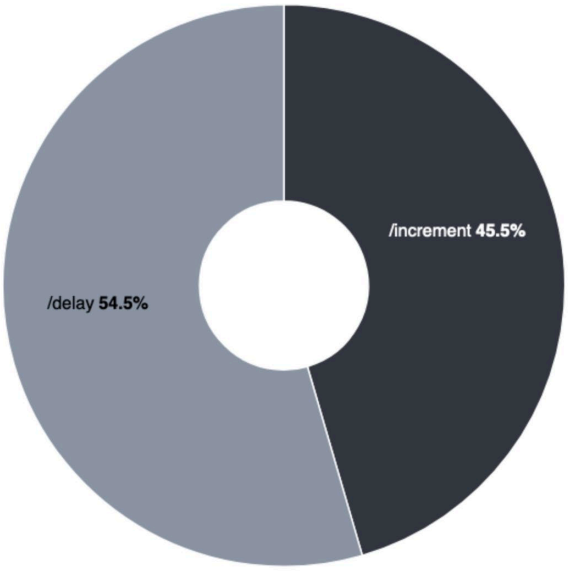
Končno točko za prištevanje zahtevka k števcu zahtevkov smo poimenovali */increment*. Klicati jo je potrebno z metodo *POST*. Uporabnik jo pokliče, kadar izvede zahtevek na neko spletno storitev. Zraven klica na to končno točko mora uporabnik podati parameter, ki pove točen čas izvedbe zahtevka. S tem sistem ve, kateri števec koraka mora prišteti.

Končno točko za izračun naslednjega prostega okna za izvedbo zahtevka smo poimenovali ***GET /delay***. Uporabnik jo pokliče z metodo ***GET***, preden želi zahtevek izvesti. Kot odgovor dobi točen čas naslednjega prostega okna za izvedbo zahtevka. Če ta čas ni v prihodnosti to pomeni, da lahko zahtevek izvede takoj.

5 Rezultati

Programsko rešitev smo po implementaciji testirali lokalno. Testi so bili uspešni, vendar je za realne podatke potrebno večjo število zahtevkov. V nadaljevanju smo predstavili nekaj rezultatov strežnika v polni obremenjenosti v produkcijskem okolju podjetja Databox [3].

Za vse zahtevke na naš strežnik smo beležili odzivni čas in s pomočjo brezplačnega orodja Kibana [4] to tudi vizualizirali. Rezultate smo zajeli za obdobje sedmih dni, natančneje od 14. 7. 2022 do vključno 20. 7. 2022. V tem času smo na strežnik naredili približno **311,43 milijonov zahtevkov**, kar je povprečno **44,49 milijonov na dan**.
Zanimivo je tudi razmerje zahtevkov med obema končnima točkama (/delay in /increment). Pričakovano je zahtevkov na končno točko za izračun naslednjega prostega okna za izvedbo zahtevka več. Namreč, če je potrebno z zahtevkom zaradi visoke porabe omejitve počakati dlje časa, smo končno točko za izračun prostega okna poklicali večkrat. Razmerje lahko podrobneje vidimo na sliki 10.



Slika 10: Razmerje zahtevkov med končnima točkama.
Vir: lasten

Prav tako smo raziskali odzivni čas strežnika na posamezni končni točki. Zavedali smo se, da je povprečen odzivni čas v nekaterih primerih slabo merilo delovanja, saj lahko nekaj počasnih zahtevkov v ključnih trenutkih pokvari uporabniško izkušnjo. V ta namen smo z zabeleženimi odzivnimi časi, z uporabo prej omenjenega orodja, izračunali še nekaj percentilnih vrednosti. Izbrali smo 95. in 99. percentil. Rezultati so prikazani v tabeli 2.

Tabela 2: Odzivni časi na posamezni končni točki.

	Povprečje (ms)	95. percentil (ms)	99. percentil (ms)
/delay	4	7	12
/increment	3	7	11

Razlika v odzivnem času je pričakovana, saj je pri izvedbi slednjega potrebno več procesiranja. Za lažje razumevanje smo enega izmed zgornjih podatkov opremili še s kratko razlago. 95. percentil na končni točki za izračun naslednjega prostega okna za izvedbo zahtevka /delay pomeni, da se 95 odstotkov vseh zahtevkov izvede v največ 4 ms. Velika večina zahtevkov se torej izvede zelo hitro.

6 Zaključek

V tem članku smo razdelali nekaj teorije metod omejevanja frekvence zahtevkov in predstavili načrtovanje ter implementacijo tega sistema na odjemalcu. Želeli smo zagotoviti enostavno uporabo in s tem dobro uporabniško izkušnjo uporabnikov in razvijalcev, ki to programsko rešitev uporabljajo. Implementacija v obliki strežnika, na katerega izvajamo zahtevke, omogoča uporabo iz kateregakoli programskega jezika. Programsko rešitev smo preizkusili tudi v realnem produkcijskem okolju in podali performančne rezultate. Percentili, ki smo jih podali, pa so nam pokazali še nekoliko jasnejšo sliko. Naša programska rešitev ima še vedno nekaj prostora za izboljšave. Predvidevamo, da je večina zahtevkov z višjim odzivnim časom povezana s sistemskimi viri na strežniku v času izvedbe zahtevkov, v nekaterih pa se najverjetneje skriva tudi kakšna napaka v naši kodi.

Literatura

[1] Rate-limiting strategies and techniques, https://cloud.google.com/architecture/rate-limiting-strategies-techniques, obiskano 21. 7. 2022.

[2] Guanlan Dai, How to Design a Scalable Rate Limiting Algorithm, https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm, obiskano 22. 7. 2022.

[3] Databox, https://databox.com, obiskano 29. 7. 2022.

[4] Kibana, https://www.elastic.co/kibana, obiskano 29. 7. 2022.

[5] Aljaž Mislovič, Sistem za omejevanje frekvence zahtevkov na odjemalcu, diplomsko delo, 2021.