University of Ljubljana
Faculty of Computer and Information Science

Matej Šnuderl

Rate limiting in API management

BACHELOR THESIS

UNIVERSITY STUDY PROGRAMME
FIRST CYCLE
COMPUTER AND INFORMATION SCIENCE

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2018

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Šnuderl

Omejevanje dostopa pri obvladovanju API-jev

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Thesis subject:

With the increasing employment of microservices in software development process much of the burden for performance assurance has moved to the web API providers. A simple approach to online API access throttling involves the access for unregistered and registered users. If we want to further restrict access at the level of various metrics (e.g. number of requests in a given time unit etc.) it is necessary to introduce a middleware layer that implements the aforementioned functionality. The thesis should highlight the scope of the Web API throttling in terms of algorithms and integrated solutions. Finally the critical evaluation should be performed on a real case study.

Tematika naloge:

Z vedno večjim vključevanjem mikrostoritev v proces razvoj programske opreme se je velik del bremena pri zagotavljanju zmogljivosti premaknil na stran ponudnikov spletnih storitev. Enostaven pristop pri omejevanju dostopa do spletnih storitev vključuje dostop neregistriranih in registriranih uporabnikov. Če želimo dostop dodatno omejevati na ravni različnih metrik (npr. število dostopov v časovni enoti ipd.) je potrebno vpeljati vmesni sloj, ki implementira omenjeno funkcionalnost. V okviru diplomskega dela raziščite področje omejevanja dostopa do spletnih virov, z vidika algoritmov in celostnih rešitev, ter na realnem primeru kritično ovrednotite izbrano implementacijo.

Za izdelavo diplomske naloge se zahvaljujem mentorju doc. dr. Dejanu Lavbiču za pomoč in vodenje pri izdelavi. Prav tako se zahvaljujem svoji družini za podporo v času študija.

Contents

List of used abbreviations

Abbreviaton	**English**	**Slovensko**
API	Application Programming Interface	aplikacijski programski vmesnik
DoS	Denial of Service	Napad za zarnitev storitve
HTTP	Hypertext transfer protocol	Protokol za transport spletnega besedila
OSI	Open Systems Interconnection model	ISO/OSI referenčni model
OS	Operating system	Operacijski sistem
RFC	Request for Comments	Zahtevek po komentarjih
POST	Request method supported by HTTP	Tip spletnega zahtevka
O(n)	Big O notation	Notacija veliki O
ttl	Time to live	Čas obstoja
MB	Megabyte	Megabajt
req	Request	Zahtevek
XML	Extensible Markup Language	Razširljiv sistem za urejevanje tekstovnih datotek
FAQ	Frequently asked questions	Pogosto postavljena vprašanja
SWOT	Strengths, Weaknesses, Opportunities and Threats	Prednosti, Slabosti, Priložnosti in Grožnje
DBMS	Database management system	Sistem za upravljanje podatkov
JSON	JavaScript Object Notation	JavaScript notacija objektov
AWS	Amazon Web Services	Amazonove spletne storitve

Abstract

Title: Rate limiting in API management

Author: Matej Šnuderl

With ever growing usage of World Wide Web, number of requests to web APIs is increasing rapidly. DoS attacks and service abuses are becoming easier to execute, and more common every day. Quality of service is becoming more important as competition is rising. To build robust and reliable services, software engineers have to take this into account when designing web APIs, to deliver end users with a pleasant and reliable experience. In this thesis we delve into rate limiting in web API management to deal with those problems on scale. We propose an approach to rate limiting when request weighting is key, and cannot be estimated/calculated upfront. We show how integration of such approach into a real working system can help in achieving high stability and performance improvements, while unlocking some advanced API monetisation opportunities.

Keywords: Rate Limiting, API management, scalable web services.

Povzetek

Naslov: Omejevanje dostopa pri obvladovanju API-jev

Avtor: Matej Šnuderl

Strma rast uporabe svetovnega spleta je silovito povečala število spletnih zahtevkov, ki jih morajo procesirati zaledni sistemi. Napadi za zavrnitev storitev in zlorabe le-teh so vse bolj pogosti in enostavni za izvedbo. Kvaliteta in zanesljivost sistemov sta ključnega pomena za ohranjanje konkurenčnosti. Naloga razvijalcev programske opreme je, da z upoštevanjem teh zahtev načrtujejo robustne sisteme, ki bodo uporabnikom omogočili prijetno in zanesljivo uporabniško izkušnjo. V tej diplomski nalogi raziščemo pristop omejevanja dostopa pri obvladovanju API-jev za reševanje omenjenih problemov. Predlagamo pristop pri katerem je obteževanje spletnih zahtevkov ključnega pomena in ne more biti ocenjeno/izračunano pred procesiranjem zahtevka. Pokažemo kako lahko integracija takšnega pristopa v delujoč sistem občutno izboljša stabilnost in učinkovitost storitev ter odpre možnosti za nove načine trženja API-jev.

Ključne besede: Omejevanje dostopa, obvladovanje APIjev, skalabilni spletni sistemi.

Daljši povzetek

Spletni API-ji postajajo ključen del velike večine spletnih storitev. Vse več sistemov se sooča s problemi skalabilnosti in zanesljivosti. Omejevanje dostopa pri obvladovanju API-jev je učinkovit pristop za soočanje s takšnimi problemi. Uporablja se za nadzor prometa v in izven sistemov. Namen tega pristopa je zmanjševanje nekontroliranih izbruhov spletnih zahtevkov v sistemu, izboljšanje zanesljivosti in stabilnosti sistema ter zmanjšanje povprečnih odzivnih časov sistema. To dosežemo s pošiljanjem prometa skozi filter/algoritem, npr. Token Bucket filter [1]. Integracija omenjenega pristopa v distribuirane sisteme predstavlja težak tehnični problem [2, 3, 4]. Kljub temu je ideja omejevanja dostopa zelo abstraktna in izvedljiva na večih nivojih, npr. na (4.) omrežni plasti skozi omrežni krmilnik, na (5.) aplikacijski plasti ISO/OSI referenčnega modela ali v OS jedru kot del nadzornika v virtualnem okolju [1].

Delovanje pristopa omejevanja dostopa

V sistemu z integriranim omejevanjem dostopa je vsakemu uporabniku storitev dodeljeno pravilo, ki določa hitrost in količino zahtevkov, ki jih sistem tolerira. Lastnik sistema se zavezuje, da bo zahtevke procesiral v razumljivem času, dokler se uporabnik drži teh pravil in njihovih omejitev. Če uporabnik prekorači število zahtevkov v določenem časovnem intervalu, so njegovi zahtevki blokirani s HTTP 429 (Preveč zahtevkov) odgovorom. Za povečanje števila in količine števila spletnih zahtevkov, je v večini primerov potrebna nadgradnja računa/naročnine. To lahko zelo pozitivno vpliva na tok prihodkov v podjetju.

Pristopi k omejevanju dostopa

Tradicionalen pristop k omejevanju dostopa predvideva, da je zahtevnost / teža spletnega zahtevka lahko določljiva/izračunana vnaprej. V teh primerih lahko omejevanje v celoti opravimo že pred procesiranjem zahtevka. Določanje teže spletnega zahtevka pa je lahko v nekaterih primerih zelo zahtevno in znatno vpliva na latenco ter hitrost odgovorov sistema. To je še posebej problematično v sistemih, ki procesirajo poljubne uporabniške skripte (programsko kodo). Določanje cene takšnih zahtevkov je zelo zahtevno, kot je zelo težko določiti tudi ali se bo procesiranje takšnih zahtevkov sploh kdaj zaključilo [5]. V poglavju 3 opišemo alternativni pristop, ki rešuje omenjen problem in je bil v našem okolju (opisanem v poglavju 7.1) ključen za pravilno delovanje omejevanja zahtevkov.

Algoritmi za omejevanje dostopa

Izbira algoritma za omejevanje dostopa je lahko odločilnega pomena pri omejevanju dostopa. Ideja omejevanja dostopa je zelo preprosta. Enostavna implementacija bi hranila le števec in ga povečala ob vsakem spletnem zahtevku uporabnika. Števec bi enostavno resetirala ob koncu vsakega intervala (vsako sekundo/uro/dan). To deluje, vendar ima pomankljivosti ki jih podrobneje predstavimo v odstavku 4.3. V tem poglavju predstavimo tudi najsodobnejše algoritme za omejevanje dostopa:

1. Token Bucket algoritem

2. Fixed Window algoritem

3. Sliding Window algoritem

Hramba podatkov

Tip podatkovne baze za hrambo podatkov o omejevanju močno vpliva na performanco omejevanja dostopa. V primeru omejevanja dostopa se soočamo s problemom zelo pogostega branja in pisanja v podatkovno bazo. Ker želimo to opraviti čim hitreje, moramo te podatke hraniti na pravilen način. Za ta namen se v večini primerov uporabljajo NoSQL podatkovne baze ključ-vrednost, ki so nastale na podlagi zahtev po visokih učinkovitostih in skalabilnosti v okolju, kakršno je svetovni splet [6].

Zaključek

Omejevanje dostopa do API-jev lahko zelo pozitivno vpliva na stabilnost in zanesljivost spletnih storitev. V našem primeru (opisanem v odstavku 7.1) smo opazili precejšno zmanjšanje količine izbruhov v spletnem prometu. Prav tako smo prepričili vse možnosti za zlorabo storitev in potencilanih napadov za zavrnitev naših storitev (DoS). Takšne sumljive izbruhe lahko sedaj kontroliramo, kar je znatno znižalo potrebe po infrastrukturi in stoške povezane z njo. S pristopom omenjenim v poglavju 3, smo lahko spletne zahtevke natančno utežili, ne da bi povečali povprečni odzivni čas sistema. V nekateih primerih smo opazili zmanjšanje povprečnega časa za odgovor do 20%. Implementacija omejevanja dostopa nam je omogočila tudi ustrezno trženje API-ja in močno dvignila število plačljivih uporabnikov naših storitev.

Chapter 1

Introduction

Web APIs are becoming essential part of vast majority of web services. Daily, more and more web APIs are facing scalability and reliability problems. Rate limiting is an important approach to web API management and is used to control amount of traffic from and into the system. Its main objective is to reduce burstiness, improve system's fairness, reliability, and stability while reducing average response times and latency. This is usually achieved by passing traffic through some filter/algorithm e.g. Token Bucket filter [1]. Integration of such mechanism into distributed environments presents a challenging technical problem [2, 3, 4].
Idea of rate limiting is very abstract and can be applied on various levels, e.g. on network (4th) layer via network interface controller, on application (5th) layer of the OSI reference model [7] or in OS Kernel, as part of the hypervisor in a virtualised environment [1]. While all levels share similar concepts, algorithms and abstractions, focus of this thesis will be on application/software layer. Software rate limiters are more flexible (i.e., can boot up at any server), scalable (i.e., multiple instances can be created for different tenants), and have more functionalities (e.g., hierarchical rate limiting) [8].
Any outage of web APIs might result into degraded user experience and dissatisfied customers. Rate limiting comes as one of the measures web API designers should take into account when designing reliable and robust web

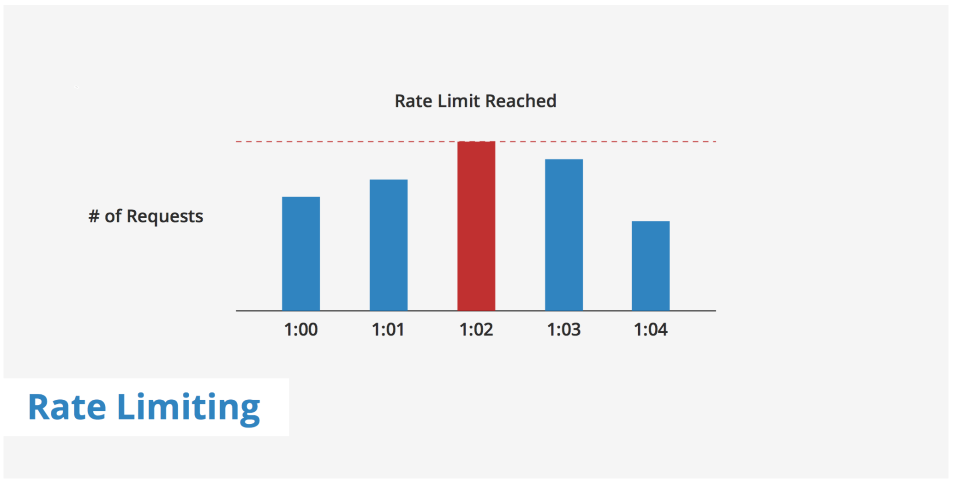


Figure 1.1: Request are grouped into 1 minute time windows (sampling periods). User requests in time window 1:02 reached the agreed upon rate limit so all further requests were blocked.

applications. Moreover, it can facilitate options for advanced API monetisation as a business model which can significantly impact revenues.
Inside of a system with web API management and integrated rate limiting each API consumer is granted a policy that states at what rate requests can be issued. As long as the consumer conforms to the agreed upon policy, the carrier promises to deliver responses in timely fashion. If the number of requests that consumer makes in given time interval exceeds the limits, requests will be blocked with a HTTP 429 (Too Many Requests - defined in RFC 6585 [9]) response. Concept can be seen on Figure 1.1. To obtain larger limits, some kind of account/subscription upgrade is usually required.

1.1 Motivation

Main motivation for this thesis is my interest in scalable and robust web services. Moreover, Sinergise (company where I am employed) required a highly customisable and flexible rate limiting implementation that could handle millions of resource expensive requests per day. This is required due to the nature and details of our web API - typical web request is treated a bit differently in our context. What does a request mean for us is described in our FAQ [10]. New approach introduced in chapter 3 was required in our case. Moreover, we wanted to utilise information from our rate limiter for load balancing. Benefits of doing this are described in section 6.4. Last but not least, our pricing policy has been per request basis for a long time, but we haven't actually had a system in place for that. Constant service abuses and exploits were a result of that.

There is also a deficiency of literature about software rate limiting as it has become a trending research problem just recently. More and more companies are facing scalability problems in their cloud based web applications and are looking for ways to monetise their APIs. Those problems can be vastly removed with integration of rate limiting. This thesis should serve as a reference when considering rate limiting and its implementation. I believe topics covered and improvements proposed will be beneficial as they area a result from an integration into an existing web API serving millions of users and tackling petabytes of data.

1.2 Thesis goals

Main goal of this thesis is to integrate rate limiting into an existing cloud based system serving millions of requests per day to improve its scalability, robustness and facilitate options for API monetisation. We want to achieve this without users noticing any degradation or downtime. To achieve this, detailed and careful analysis of the field is required. Another goal is to perform a detailed comparison between current "state of the art" algorithms for rate limiting and propose a few changes and improvements. Last but not least, we want to present our approach and results of our integration and its positive impacts on our system's working.

1.3 Related work

In this section we briefly present related work to rate limiting in web API management. Existing solutions are implemented either as a middleware, or as an API Gateway.

1.3.1 Middlewares

There are existing open-source rate limiting implementations in shape of middlewares for languages like Go [11] or Node.js (Javascript runtime) [12]. There is no notable differences amongst those except the implementation selection which is restricted to programming language of the service they are being injected into. Every request arriving into the system, goes through the applied middleware (see figure 1.2) before usual request processing occurs. Because they work on service level, using in memory storage, state cannot be shared across different nodes in your cluster out of the box. Moreover, no additional tooling for API management is provided like in the case of API Gateways. Middleware can be precisely customised, but a lot of implementation details and deep understanding of rate limiting is still needed to integrate such middleware into a system.

1.3.2 API Gateways

Full-fledged solutions are usually implemented as API Gateways (seen on figure 1.3) that are positioned in front of your services and can be hosted in cloud or on premise. API Gateways can work completely separately from your system. Every request goes through them before reaching the system. Functionalities like caching, rate limiting, authentication and more are usually provided out of the box or as plugins. Amongst others, some big players in this field are Kong [13], Wso2 [14], Zuul [15], Nginx API Gateway [16], API Umbrella [17], and Tyk [18]. Those are industry tested and come with all the tooling to efficiently manage your web APIs. As existing solutions using this approach work on bigger scale, they are less flexible and customisable.

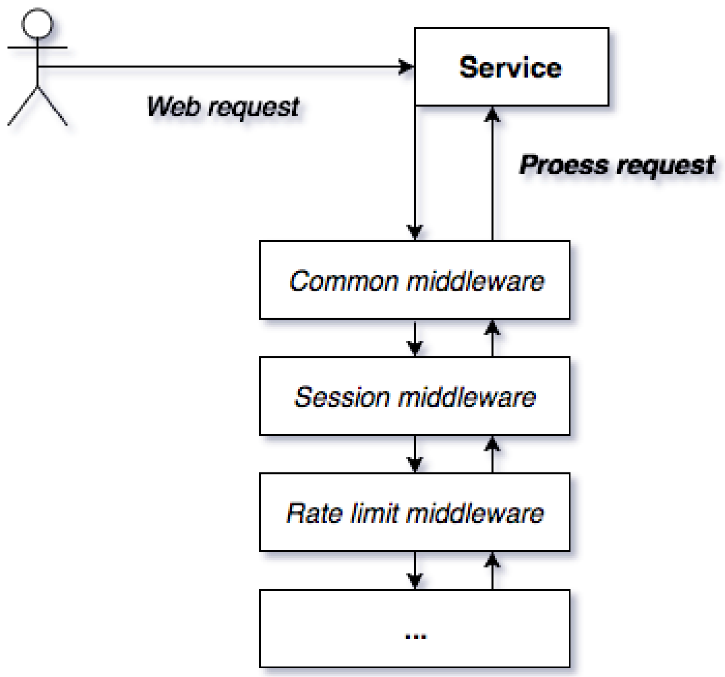


Figure 1.2: Service middleware architecture.

Customisation is usually done through configuration files and/or provided API endpoints.

These solutions try to implement everything out of the box which works great in majority of cases but comes at its own expenses. We found existing solutions too heavyweight with lack of lack of detail to the thing we cared about - performant rate limiting. They may introduce increased latencies that are not acceptable in environments with tight time constraints. Moreover, more precise, advanced - on domain level configuration is sometimes required which cannot be expressed through configuration files or would require a lot of effort and/or code modifications. Such attempts usually result into creating your own fork of the open-source implementation which later proves to be a nightmare to maintain. API Gateways mentioned also follows the traditional approach explained in chapter 2. As such, they are not suit-

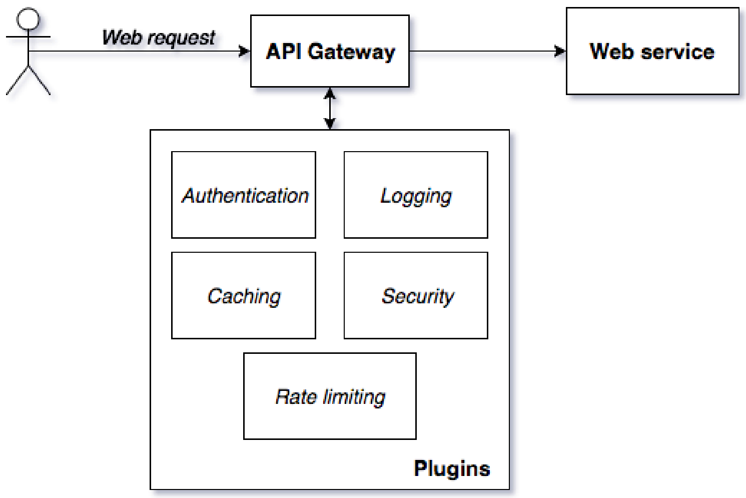


Figure 1.3: API Gateway architecture.

able for some systems and custom domain driven implementation is required.

1.4 Thesis overview

In chapter 2 we explain key problems of traditional approach to rate limiting and its limitations in certain situations.

Chapter 3 proposes an alternative approach to rate limiting with focus on environments in which request weight cannot be estimated/calculated upfront.

In chapter 4 we look through current state of the art algorithms for rate limiting. We analyse their benefits, drawbacks and complexities they bring with the implementation. In section 4.5 we propose a few improvements over algorithms traditionally used.

Chapter 5 describes different approaches for database storage of rate limiting data. In particular, we discuss centralised and distributed database approach and advanteges of NoSQL over SQL databases in context of rate limiting.

Chapter 6 discusses role, importance and approaches to efficient API management and tooling around it when integrating rate limiting.

Chapter 7 presents the results and describes what integration of rate limiting provided us with and how that affected our system.

Chapter 2

Traditional approach

Traditional rate limiting implementations assume that weight of request is known/can be estimated upfront. In such cases rate limiting is done prior to the request being processed as seen on figure 2.1. This is not always the case or can introduce increased latency and significant overhead to response times.

Problem

Imagine we had a web API resource endpoint that executed custom code - user script sent through a POST request. Code could be arbitrary and look something like:

```
var x = 1;
```

Listing 2.1: Simple custom script.

or

```
function fibonacci(num) {
  if (num <= 1) return 1;
  return fibonacci(num − 1) + fibonacci(num − 2);
}

fibonacci(10000);
```

Listing 2.2: Expensive custom script.

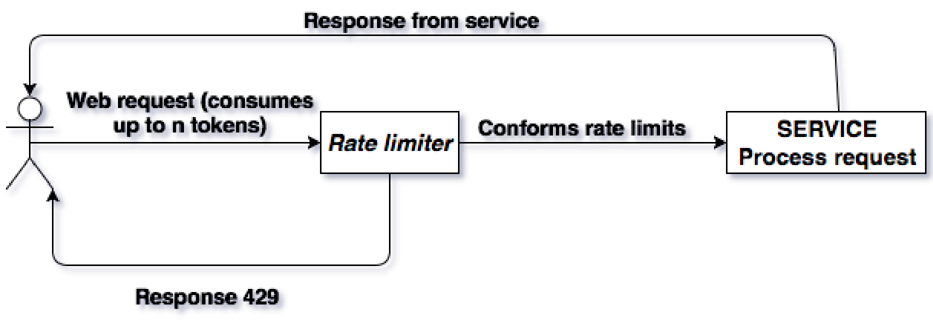


Figure 2.1: Traditional approach to rate limiting - request weight is known upfront, prior to the request processing.

While first script has time complexity of $O(1)$, second has a time complexity of $O(2^n)$. There is a significant difference in the compute power we require in order to execute those scripts. It would be unfair to treat both requests with same weight in the context of rate limiting. A few of expensive user scripts could do more harm than a million of simple ones. It is impossible to calculate/estimate weight of such request upfront. Due to undecidability of the halting problem we cannot even tell if the user's script will ever finish, let alone estimate its running time [5].

Similarly, request weight estimation can be very expensive (include complex computations) and introduce large overheads with increased response times. Problem is best seen on line 2 of the following listing.

```
1 def expensive_request(request):
2     weight = calculateRequestWeight(request) # This can be very expensive to calculate - user
          has to wait for response!
3     if rate_limiter.conformsRateLimits(weight):
4         return process_request(request)
5     else:
6         return TOO_MANY_REQUESTS
```

Listing 2.3: Estimating request weight before request processing may be very expensive, hence increasing response times.

Integration of traditional approach may have more drawbacks than benefits in such scenarios. Users that rely on quick response times will have to wait longer for responses. We cannot afford that, as one of the goals of rate limiting is to reduce average response times and latency. We propose an alternative approach that takes such cases into consideration in the following chapter.

Chapter 3

Proposed approach

To reduce response time overhead and latency introduced by request weight estimation, we propose an approach with post request weight correction. This approach extends the traditional one and corrects the request weight after request have been processed. At that point, user has already received his response and we have all the information about the request (its duration, calculations performed, etc.) to accurately weight it without introducing response time overhead. This is illustrated on figure 3.1. We still have to do traditional pre-processing rate limiting to avoid dangerous windows. Were we not doing that, user could do infinite number of requests in those windows until some response corrections took place. Post request weight correction solves problems outlined in chapter 2 but at a cost of a few regressions. In following sections we use token count as number of requests user can issue before hitting the limits. Request weight indicates how many tokens should he consumed.

3.1 Extended bans

Doing post request weight correction may lead to extended bans for API users. This happens when a user does a request that is weighted more than his current token count. Token count might become negative thus extending user's ban. Until positive token count is accumulated, no further request can be done. If this is not an issue, this can actually be extremely valuable as it prevents several kind of service abuses.

3.2 Additional storage roundtrip

To correctly update weight after request took place we need to do an additional roundtrip to our storage to fetch the updated token count. Imagine user with 5 tokens left issuing a request to web API. We would consume 1 token before the request and consume others with the actual request weight afterwards. Assuming request is worth 3 tokens, we have to consume 2 more tokens (we did consume 1 initially). Normally, we would expect token count after the request to be 2. However, token count state in our storage might change during the lifespan of this request. Other requests might took place and consume the tokens. Similarly, new token distribution could occur and update token count. Have we not taken these cases into account, we would override those events and introduce huge flaw into our rate limiting implementation. Additional storage roundtrip can be avoided with databases that supports atomic decrement operations or multi operation transactions. In those cases, storage implementation takes care of decrementing the value currently stored or is able to perform multiple operations in an atomic fashion to avoid race conditions.

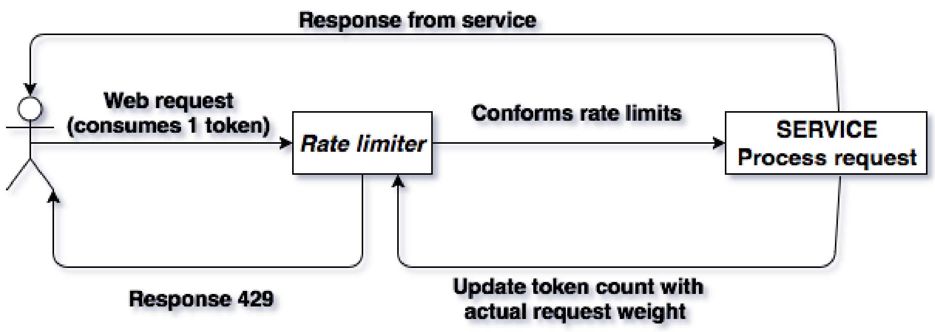


Figure 3.1: Rate limiting with post request weight correction.

Chapter 4

Rate Limiting algorithms

4.1 Overview

Idea of rate limiting is very abstract and simple. It can be implemented in numerous ways which led to evolution of a few "state-of-the-art" algorithms. They are all trying to achieve the same goal, each with its own drawbacks. Properties that are common to all:

1. Sampling period
2. Some kind of a counter to keep track of the request count

We will use requests instead of bytes/packets (mostly used in literature [19, 20, 21]) as a unit of measurement over next sections. Same reasoning applies to both if we use request weighting (some requests may be more expensive than others). Algorithms will be compared without the notion of queues. Queues are needed in network rate limiters as they have to process all of the incoming traffic. This significantly complicates rate limiting implementations and may increase latencies through queue congestions. Interesting approach with queue jumping was recently introduced that attains near-ideal performance [22]. Queues are not required in software rate limiters as we can simply discard violating web requests (block with a HTTP

429 response). This also removes additional layer of complexity of the following algorithms. What we call sampling period in coming parts is a unit of measurement for our counting rules, e.g. second in 10 req/sec, hour in 1000 req/hour,...
Most naive implementation of rate limiting would be to simply increment a counter with every request and reset it with a start of a new sampling period. This works but has some disadvantages as discussed in section 4.3. We will use the following metrics to assess algorithms in the following sections:

1. Memory footprint
2. Accuracy / atomicity (ability to update the state with single operation during which no other operation might modify the state)
3. Consistent distribution of traffic

It is important to consider all of those when choosing the appropriate algorithm. Differences might look small but add up quickly in web systems with high traffic volumes.

4.2 Token bucket algorithm

Token bucket algorithm is very simple and flexible. It is heavily used in telecommunications networks [3] but can also be used for purposes like scheduling. It is based on an analogy of a fixed capacity bucket into which tokens are added at a fixed rate. The algorithm is liked for its simplicity which is nicely illustrated on figure 4.1.

Bucket has a defined maximum capacity/depth of size n and a defined rate r at which tokens are being added. Bucket might overflow due to its fixed capacity. This is good for API provider as it prevents tokens to accumulate when consumption is idle. Token accumulation could lead to spikes and bursts in consumption which we would like to avoid.

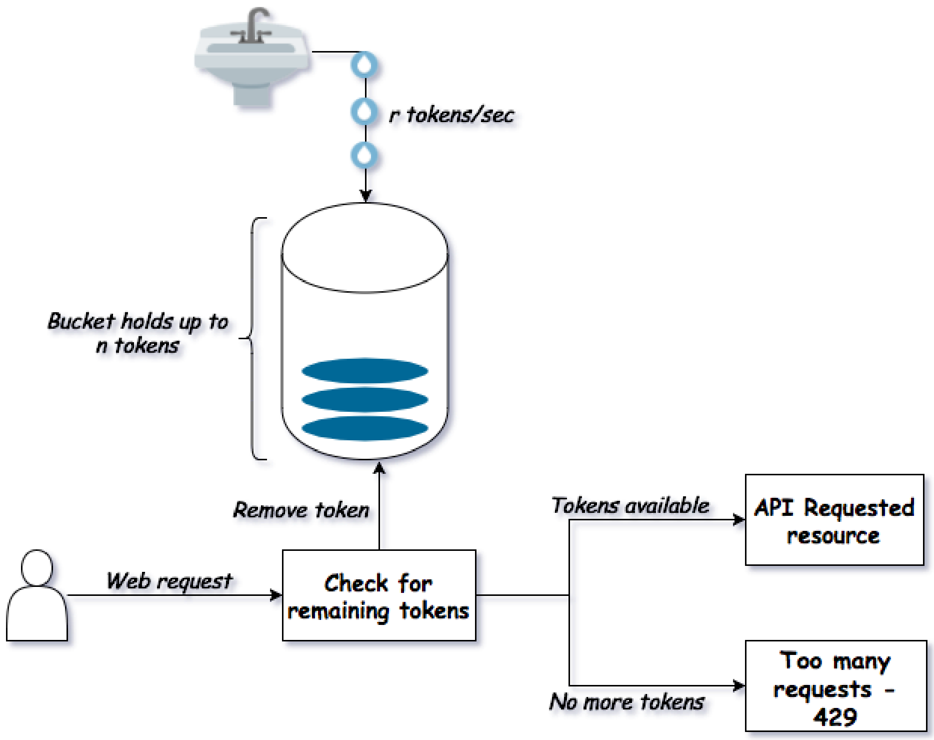


Figure 4.1: Visual presentation of the token bucket algorithm. Tokens are being added into a bucket at fixed rate. Each packet/request going through network consumes a token. If no token is available, request is denied.

For every user request, algorithm checks for number of tokens currently in the bucket. If there are less tokens in the bucket than request weights, rate limit has been exceeded resulting in a denied request. If the request conforms to all of the policies it is passed through to the requested resource. Token bucket algorithm is very flexible and never loses/overflows any of the data [21].

4.2.1 Memory footprint

For each user we have to store timestamp of the last token distribution and current token count in the bucket. We can store timestamp in a 4-byte

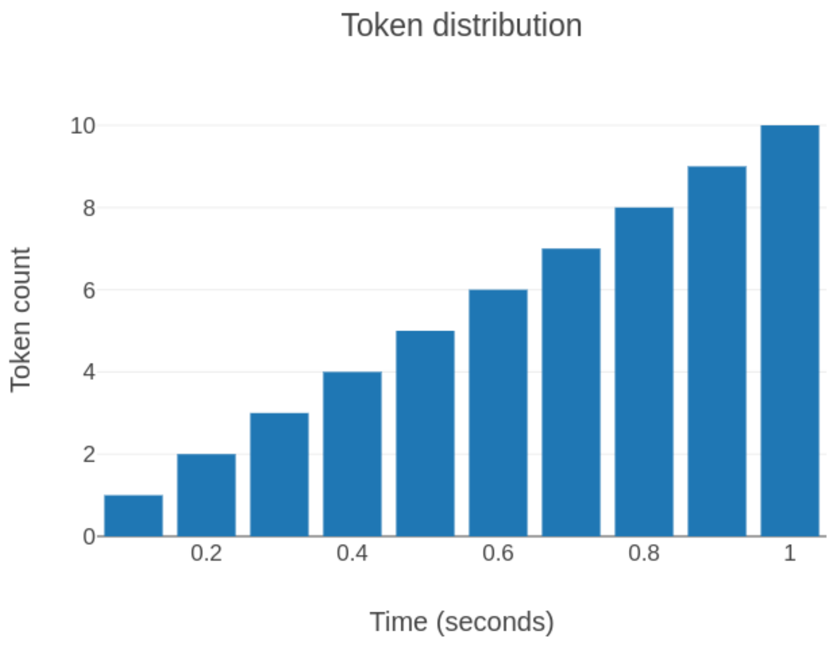


Figure 4.2: Token bucket algorithm provides consistent token distribution without spikes. Figure displays token distribution for a refill policy of 1 request per 100 milliseconds.

integer and token count into 2-byte short (should suffice for most of the use cases) resulting in a tiny memory footprint of a total 6 bytes per user. With 1 million active users this would sum into 6MB of space in total.

4.2.2 Accuracy/atomicity

Token bucket algorithm lacks atomicity due to its ”read-and-then-write” nature. For each request we have to first read the record to distribute new tokens. This can lead to race conditions in distributed environments resulting in some unconformant requests passing through the rate limiter. Imagine there was only one token left in the bucket. If user issued multiple requests and requests were served by multiple processes that would read the token count simultaneously before either of them updating it, each process would think user still has a token left and thus not hitting the limits. This could be solved with locks/transactions/entry processors, but that has to be handled by the storage.

4.2.3 Consistent distribution of traffic

Token bucket algorithm has consistent token distribution as seen on figure 4.2. It prevents any kind of sudden bursts or spikes in traffic. However, tokens in bucket may accumulate which can lead to double the amount of expected requests done in one sampling period. For example, user with rate limit policy of 10reqs/s might have accumulated full bucket (as a result of not doing any requests in the previous sampling period). As soon as he starts issuing the requests, he will consume tokens. Simultaneously, new tokens will be added to the bucket. Those are added consistently, so the issue is not as problematic as with the fixed window algorithm, but should be taken into account.

4.2.4 Example implementation

```
import time
from django.http import HttpResponse

def distribute_new_tokens(current_timestamp, last_distribution_ts):
    time_elapsed = current_timestamp - last_distribution_ts
    num_new_tokens = int(time_elapsed / ADD_RATE)
    return num_new_tokens

def token_bucket_view_handler(request):
    current_timestamp = time.time()
    user_rate_limits = database_handler.get(user_identifier)
    last_distribution_ts = user_rate_limits.get("last_distribution_ts", current_timestamp)

    num_new_tokens = distribute_new_tokens(current_timestamp, last_distribution_ts)
    n_tokens = user_rate_limits.get("tokens", BUCKET_CAPACITY) + num_new_tokens
    n_tokens = min(n_tokens, BUCKET_CAPACITY)

    if n_tokens == 0:
        return HttpResponse("Too many requests", status=429)
    else:
```

```
    database_handler[user_identifier] = {"tokens": n_tokens - 1, "last_distribution_ts":
   current_timestamp}
    return HttpResponse(f"Num requests left: {n_tokens - 1}")
```

Listing 4.1: Example token bucket algorithm implementation in Python.

4.2.5 Practical considerations

It comes as no surprise that token bucket algorithm is so widely used. Tiny memory footprint, simplicity and consistent traffic distribution makes it a perfect candidate for most of the use cases. While implementation is very simple, token distribution calculations might be problematic for extremely short distribution intervals (sub milliseconds). This is due to the clock synchronisation/accuracy problems that are very common in distributed systems [23]. It turns out that such accuracy is rarely needed so this can be ignored most of the times.

4.3 Fixed window algorithm

Concept of fixed windows can be used for many purposes including rate limiting. To work, counter for user's requests in a current rate limit's window of length t has to be kept. Initial counter starts at 0. If user exceeds number of requests stated by the policy in a current window, access is denied. Imagine we have a sampling period of 1 day. All requests sent throughout the day would fall into the same window. Start of a new day would mean a fresh start with all counters being reset to 0. This is nicely illustrated on figure 4.3.

4.3.1 Memory footprint

Memory footprint for fixed window algorithm depends on the implementation. We need to store at least one 2-byte short for current window's request count per user. In this case, window bounds (start and end) have to be kept by supervisor system and reused by all of the policies. For more flexible

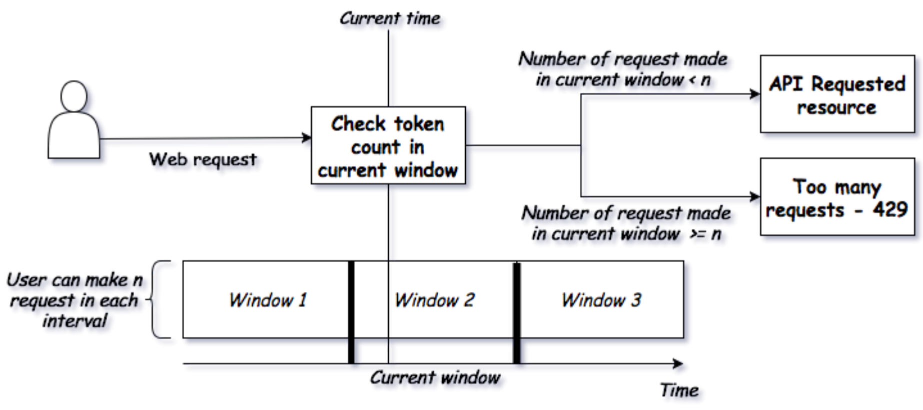

Current time
Number of request made in current window < n
API Requested resource
Check token count in current window
Web request
Too many requests - 429
Number of request made in current window >= n
User can make n request in each interval
Window 1
Window 2
Window 3
Current window
Time

Figure 4.3: Visual presentation of fixed window algorithm.

version of the algorithm, window bounds should be kept by each policy. As a matter of fact we only have to store one window bound, as other can always be calculated from the window length. This increases memory footprint per user from 2 bytes to 6 bytes but increases flexibility and accuracy of the algorithm.

4.3.2 Accuracy / Atomicity

With fixed window algorithm we can achieve atomicity when doing rate limiting. Because of "write-and-then-read" nature we can assure correct operation of the system even in distributed environments.

4.3.3 Consistent distribution of traffic

Major drawback of fixed window approach is its inconsistent distribution of traffic. It can sometimes let through twice the number of allowed requests. This is more problematic than with token bucket algorithms as this might happen instantaneously. Tricky part are window boundaries of time intervals as seen on figure 4.4. User could consume all of his request moments before window ends and do it again as the new window starts. This can be solved

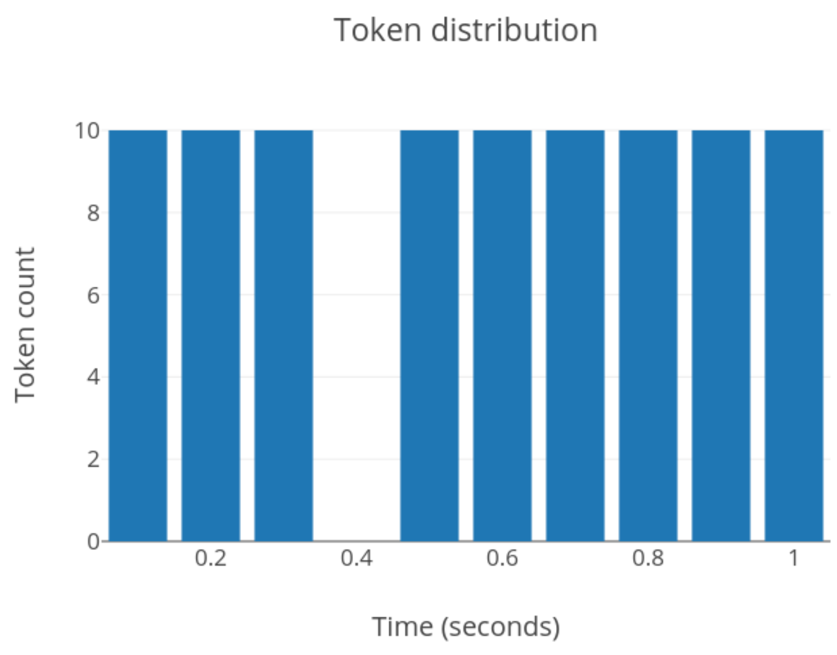


Figure 4.4: Fixed window token distribution with refill policy of 10 request per 500 milliseconds. User issued 10 requests at 0.4s into the window's length. New window starts at 0.5s and resets the counter to 0. User can issue 10 more request at 0.5s into the window and thus issue 20 requests in just 100 milliseconds. This is more than the rate limit policy states and can lead to unforeseen problems.

with smaller rate limits in between but this would enforce too severe rate limits on users and their requests. Moreover, it would additionally complicate the rate limiter implementation.

4.3.4 Example implementation

```
import time
from collections import defaultdict
from django.http import HttpResponse

class FixedWindowManager(object):
    def __init__(self, window_length=WINDOW_LENGTH):
        self.window_length = window_length
        self.window_start = int(time.time())
        self.window_end = self.window_start + self.window_length

    def move_window(self, database_handler):
        self.reset_bucket_states(database_handler)
        self.window_start = self.window_end
        self.window_end = self.window_start + self.window_length

    def get_user_bucket_key(self, user_identifier):
        return f"{user_identifier}_{self.window_start}"

    def is_out_of_window(self, request_ts):
        return request_ts >= self.window_end

    def reset_bucket_states(self, database_handler):
        database_handler = defaultdict(int)

def fixed_window_view_handler(request):
    current_timestamp = int(time.time())

    if fixed_window_manager.is_out_of_window(current_timestamp):
        fixed_window_manager.move_window(database_handler)

    user_bucket_key = fixed_window_manager.get_user_bucket_key(user_identifier)
    database_handler[user_bucket_key] += 1

    if database_handler[user_bucket_key] > NUM_REQUESTS_PER_WINDOW:
        return HttpResponse(f"Too many requests. New requests in {fixed_window_manager.window_end - current_timestamp} seconds.", status=429)
    else:
        return HttpResponse(f"Num requests left: {NUM_REQUESTS_PER_WINDOW - database_handler[user_bucket_key]}")
```

Listing 4.2: Example fixed window algorithm implementation in Python.

4.3.5 Practical considerations

Fixed window approach to rate limiting has a major drawback from its inconsistent distribution of traffic. Even though we can easily perform atomic updates in distributed environments, letting through twice as much of the requests than we would like is unacceptable in many applications. Moreover, fixed window approach is not consumer friendly after tokens have been consumed. To issue new requests, consumers have to wait until end of the entire window, which may be very long.

4.4 Sliding window log algorithm

Sliding window log algorithm extends fixed window algorithm with the notion of a moving window. Naive implementation optimizes for accuracy - it stores a timestamp for each user's request in a single sorted data structure like map or set [24]. This allows for an efficient removal of all outdated entries. To get numbers of requests in past hour, we have to sum all of the entries in our data structure. Algorithm optimises for accuracy, but suffer from large memory footprint and speed overhead.

4.4.1 Memory footprint

Sliding window log algorithm leaves a large memory footprint which might be problematic in some environments. It stores an entry for each request which in practice could mean things could get out of control rapidly. An average amount of 1.000 user request per day with 20.000 active users would result into 20.000.000 storage entries. With each stored timestamp value being 4 byte integer this would accumulate into a total of 20.000.000 * 4 bytes = 80MB.

4.4.2 Accuracy / atomicity

Algorithm implementations can achieve high accuracy and atomicity due to algorithm's "write-and-then-read" nature. This is achieved by inserting current timestamp into our data structure before reading its state. Atomic insertion is supported by most of the key-value databases.

4.4.3 Consistent distribution of traffic

In contrast to fixed window algorithm, sliding window log assure consistent and stable distribution of traffic. Requests are removed from the sorted set on the fly as they fall out of the window. Token distribution is therefore proportional to the distribution of requests issued.

4.4.4 Practical considerations

While the accuracy and simplicity of sliding window algorithm might be useful in many applications, it leaves a huge memory footprint which can lead to problems in memory limited environments. Moreover, sliding window algorithm produces one more noteworthy side effect. Namely, algorithm continues to store request entries even after user exceeds the rate limit. This may be advantageous in some situations as it extends the ban on potential abuses of API. On the other hand, it can lead to problems in case of DoS attacks. Storing entry for every incoming request can quickly lead to out of memory errors. Some kind of defensive mechanism needs to be introduced to avoid such failures.

4.4.5 Example implementation

```python
import time
from django.http import HttpResponse

def sliding_window_view_handler(request):
    curent_timestamp = time.time()
    window_start = curent_timestamp - WINDOW_LENGTH

    user_rate_limits = database_handler.get(user_identifier)
    user_rate_limits[curent_timestamp] = curent_timestamp

    requests_inside_sliding_window = {ts: ts for ts in user_rate_limits.keys() if ts >=
      window_start}

    if len(requests_inside_sliding_window) >= BUCKET_CAPACITY:
        return HttpResponse(f"Too many requests", status=429)

    database_handler[user_identifier] = requests_inside_sliding_window
    return HttpResponse(f"Num requests left {BUCKET_CAPACITY - len(
      requests_inside_sliding_window)}")
```

Listing 4.3: Example sliding window algorithm implementation in Python.

4.5 Sliding window counters algorithm

Being tempted by the high accuracy, consistent distribution of traffic and atomicity of sliding window log algorithm, we looked for an implementation

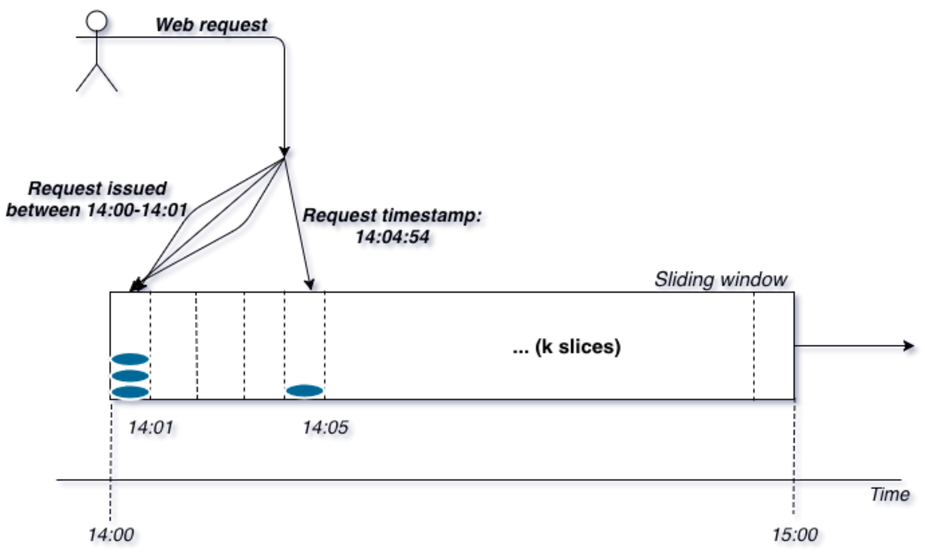


Figure 4.5: Sliding window counters algorithm. It groups requests into smaller time slices reducing its memory footprint.

that could reduce memory footprint without significant tradeoffs. We knew we had to store timestamps for sliding window approach, but we required amount of those to be constant in regards to number of requests. To achieve this we propose splitting sliding window into k intervals each of size 1/k of the original window. With sampling period of 1 hour and k=60 we get 60 time slices (buckets) of 1 minute. As requests are coming in, we round their timestamp to the start of the closest bucket. For example, user request at 13:44:58 would fall into 44th bucket - incrementing its token count. Concept is visualised on figure 4.5. Using this approach, we can control algorithm's memory footprint and consistent distribution of traffic. With k set to 1 we would get a fixed window algorithm implementation.

4.5.1 Memory footprint

Memory footprint can be controlled as needed with regulating the parameter k. With k=1 algorithm has to store one timestamp and a token count. This accumulates to (6-8) bytes per user depending on data types used. As more accuracy is needed, footprint increases proportionally to k. Memory footprint equals k * (6-8) bytes. What makes this algorithm work is it constant memory footprint regarding to k. This mitigates issues of sliding window log algorithm during sudden traffic spikes or DoS attacks.

4.5.2 Accuracy / atomicity

As with sliding window log algorithm, bucket updates can be done with atomic operations.

4.5.3 Consistent distribution of traffic

Consistent distribution of traffic is controlled through parameter k. With k=1 we face same issues than in fixed window algorithm. As the k increases, distribution of traffic becomes consistent.

4.5.4 Practical considerations

Using sliding window counters algorithm we were able to find a middle ground between memory footprint and accuracy that we required. While map data structure is required with request weighting, it can be implemented using a set when all request are weighted equally. Either way, data structure should be consistently sorted to allow for quick removals. Picking appropriate parameter k depends on your needs and your sampling period. As the sampling period increases, so should the k (to avoid too long time buckets).

4.6 Comparison

There are a few rate limiting algorithms out there that should be considered when integrating rate limiting into your system. Suitability of them depends on one's use case and environment in which they are deployed. In memory limited environments algorithm's footprint is of a huge importance while some applications require 100% accurate rate limiter. Memory footprint comparison for example load of 500req/day and 100req/day is roughly estimated in table 4.1. It is clear that sliding window log algorithm has incomparably larger memory footprint than others. Furthermore, it is important to note that sliding window log memory footprint increases proportional to the number of requests, while other implementations stay constant.

While memory footprint might be of a big importance in many environments, some APIs favours accuracy and atomicity which is crucial to their existence. Achieving perfect accuracy and atomicity in distributed environments is very tough problem to solve without large tradeoffs in performance. Consistency is another important part of the rate limiting system that should be taken into consideration. Specific properties of algorithms are summarised in table 4.2.

	Properties		
Algorithm	Atomicity	Memory footprint	Traffic consistency
Token bucket	No	Low	Medium
Fixed window	Yes	Low	Low
Sliding window log	Yes	High	High
Sliding window counters	Yes	Customisable	Customisable

Table 4.2: Overview of key properties in discussed rate limiting algorithms

	Rate limiting algorithm			
[500req/day]	Token bucket	Fixed window	Sliding window log	Sliding window counters
1.000 users	0.006MB	0.002MB - 0.006MB	2MB	Customisable
10.000 users	0.06MB	0.02MB - 0.06MB	20MB	Customisable
100.000 users	0.6MB	0.2MB - 0.6MB	200MB	Customisable
1.000.000 users	6MB	2MB - 6MB	2GB	Customisable
[1.000req/day]	—	—	—	—
1.000.000 users	6MB	2MB - 6MB	4GB	Customisable

Table 4.1: Memory footprint summarised for rate limiting algorithms

Chapter 5

Storage

Storage plays a key part in any rate limiting implementation. Records have to be stored and retrievable at any point. As they have to be read for every request entering the system, database has to be able to handle huge amount of concurrent reads and writes. Number of requests hitting the database backing rate limiter may go through the roof during DoS attacks or service abuses. To handle such loads, DBMS were designed for datasets that are accessed (often simultaneously) by many users, for both reading and writing [25]. DBMS should be able to handle such spikes reliably without failures and without significant latency penalties. In fact, it should just slow down the queries of the user violating the rate limit policies. While in memory storage is a viable option for some applications, it doesn't work in distributed environments and can't recover after system failures. Restoring rate limiting state under system failures is crucial in environments with refill policies of long sampling periods. Some kind of persistent storage is needed in such cases but that comes with its costs, e.g. snapshotting and increased latency.

Key properties for a rate limiting database:

1. **Data partitioning / sharding**: ability to shard users by their unique identifiers to improve performance
2. **Multi operation transactions**: ability to combine multiple opera-

tions into a transaction to avoid race conditions

3. **Low latency**

5.1 SQL vs NoSQL

NoSQL databases evolved from requirements for high performance and scalability in an environment such as the World Wide Web [6]. As performance is essential to more and more applications, tech leaders like Facebook and Amazon developed their our NoSQL databases like Cassandra and DynamoDB [26]. For purposes of rate limiting, key-value NoSQL databases are usually used. They benefit from even higher performance as performing a lock, join or union can be avoided when querying the data. There are many open-source key-value databases that are appropriate for the purposes of rate limiting. They come with multiple programming language support, are industry tested and employ key properties required for rate limiting listed in previous section. To name a few top contenders:

1. Redis [27, 26]
2. Hazelcast [28, 29]
3. Aerospike [30, 31]

Your team's experience with those technologies should motivate decision on suitable key-value database. Deciding on "hottest" technology and following the "hype driven develpment trend" is foolish and should be avoided. From benchmarks we did, data serialization technique contributed most (more than a key-value database technology) to performance and latency of rate limiting. Data serialization influence amount of data that has to be transferred through network. Network data transfer should be as tiny as possible since it is usually a bottleneck in distributed systems. Moreover, serialization technique also lead to reduced memory footprint and smaller state of the database.

5.2 Distributed vs centralised

There are some clear advantages using distributed database e.g. improved reliability, higher performance and scalability. However, using a distributed database introduces complexity, difficulties maintaining data integrity and eventual consistency problems [32]. As reliability and performance are key to rate limiting, we found distributed storage approach work significantly better. In centralised database, data is managed by a single DBMS on a single node. In case of a failure any data access and consequently rate limiting is impossible.

Data integrity and eventual consistency problems of decentralised databases can be solved with appropriate data partitioning. With data partitioning based on user's unique identifier, same node will always handle requests from the same user. With this approach, we have a centralised approach to user's data integrity and consistency while preserving distributed database advantages.

5.3 Do we really need persistence?

Ability to restore rate limiting database state after system failures feels very tempting. However, in large systems with numerous users database persistence can quickly become very expensive. As this is tightly related to algorithm's memory footprint, it varies significantly between implementations. Frequent snapshotting of rate limiting database is required to actually benefit from database backups. We believe this approach does not scale well and should be avoided.

It turns out that most of the rate limiting policies, those that are not business related, have sampling periods below 1 minute or even below 1 second. Persisting those don't seem very reasonable as they do not contain any business value. Losing 1 second of rate limiting data that serves just as a service protection layer seemed impractical. It is a tradeoff for huge storage saving one should be willing to take. On the other hand, we found huge value in persistence of our business related policies. In our case, those could have sampling periods to up to 31 days. Losing 31 days worth of information about user's API access might be problematic. We did our API monetisation based on that data so their integrity and persistence was crucial to our monetisation model. Moreover, persisting business policies was not expensive as their percentage was and should stay relatively small.

5.4 Practical considerations

A lot of thought should be put into deciding on suitable storage technology, going centralised or decentralised or even thinking about rate limiting persistence. System requirements may change and require different properties from the database. Architecture your system in a way that storage technology can easily be replaced and switched with any implementation effortlessly. Following this practice enabled us to rapidly benchmark different storage technologies and iterate our implementation.

Chapter 6

API management

Integration of rate limiting introduces additional layer of complexity into a system to facilitate some powerful features. As everything should work seamlessly in the backend, observability of the system in real time is very important. Management/Support shouldn't be forced to understand the underlying implementation to configure user policies in real time. They should be able to answer business related questions like why is some customer being blocked or when will he be able to do further requests. High level abstractions and tools around rate limiting implementation are required for this to be achieved.

6.1 API monetisation

Rate limiting unlocks powerful opportunities to monetise web APIs. More and more companies are leaning towards this model to drive revenues. Deep understanding of the API and its values is required to monetise it efficiently. For monetisation to be successful, value must be obtained by all participants in the API economy value chain seen on figure 6.1 [33]. On June 11, 2018, Google has changed their Maps API pricing model [34]. This raised a lot of dissatisfaction in the IT community with some customers reporting cost increases for up to twenty times through social media. A lot of their API

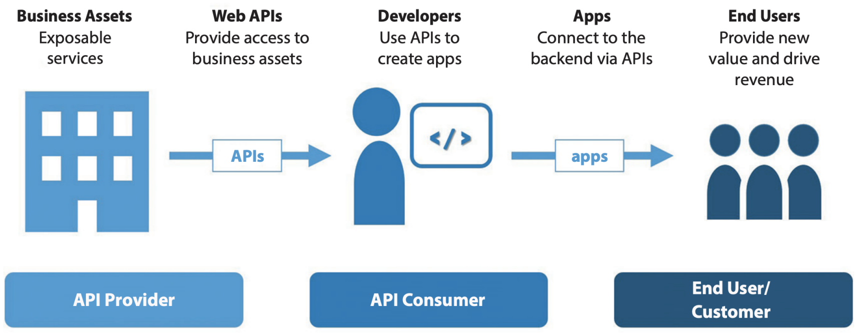


Figure 6.1: The API economy value chain. [33]

customers became unprofitable, seeking for cheaper alternatives. Any drastic change to the API pricing models should be backed by a thorough research. Any change that violate successful monetisation principle may introduce decline in revenues.

6.1.1 Rate limiting algorithm

Rate limiting algorithm can play a major role in API monetisation. Some customers might have special requirements about their API consumption. For example, they require all of the tokens available with the start of a new sampling period for quick constumption. This is achievable by fixed window algorithm discussed in section 4.3 and by sliding window counters algorithm proposed in section 4.5. On the other hand, some customers might require tokens consistently through the whole day. Locking yourself into one implementation might drive some customers away as request patterns enforced by the algorithm does not suit their needs. Having the flexibility to change implementation per user basis may benefit both API providers and API consumers.

6.2 Refill policies

Refill policies abstraction should expose exactly two properties. ”How many” in ”what time”. Everything else should be an implementation detail and hidden from customers. We can abstract a refill policy of 20 requests per hour in XML as:

```xml
<?xml version=”1.0” encoding=”UTF−8”?>
<RefillPolicy>
  <Capacity>20</Capacity>
  <SamplingPeriod>HOUR</SamplingPeriod>
</RefillPolicy>
```

Listing 6.1: Refill policy of 20 requests per hour.

Any other format, e.g. JSON can be used that that allows to combine such policies to create a contract between API consumer and API provider.

```xml
<?xml version=”1.0” encoding=”UTF−8”?>
<Contract>
  <Identifier>User1235</Identifier>

  <RefillPolicy>
    <Capacity>20</Capacity>
    <SamplingPeriod>HOUR</SamplingPeriod>
  </RefillPolicy>

  <RefillPolicy>
    <Capacity>1000</Capacity>
    <SamplingPeriod>DAY</SamplingPeriod>
  </RefillPolicy>
</Contract>
```

Listing 6.2: Rate limit contract for User1235.

Rate limit contract abstraction as seen on listing 6.2 is understandable to anyone. Contract is bound to unique user and can define arbitrary number of policies. Editing policies in XML files should be simple enough, but can still lead to errors due to human mistakes. Any syntax error in such file could corrupt it and break the contract or potentially whole system. It is convenient to edit contracts through some kind of web interface like shown on figure 6.2 and let backend expose APIs to work with the underlying implementation directly. Management/Support staff shouldn’t know details about where or how contracts are stored.

6.3 Consumer feedback

To develop consumer friendly rate limiter, one has to think about methods to provide users with feedback about their limits in real time. API consumers may depend on your system and any rate limited request may result into their system's failure. There are two reasonable approaches that helps your customers with conformant API consumption.

Response headers

Response headers approach is favoured as it enables consumers to perform programmatic backoff. Giants like Twitter and Github use headers beginning with 'X-RateLimit' or 'x-rate-limit' to indicate rate limit information. [35] [36]. This is not a standard but seems to be a common practice amongst existing software rate limiting implementations. Headers commonly seen in responses:

1. X-RateLimit-Limit
2. X-RateLimit-Remaining
3. X-RateLimit-Reset
4. Retry-After

Interpretation of headers is algorithm specific but can be briefly summarised by their respective name. API consumer can leverage those headers and develop their system around them to avoid any potential rate limits and conform to their policies.

Web interface

Exposing web interface similar to one seen on figure 6.2 looks nice and may be important to some enterprise customers that are not as technical. It may help them understand their limits in more intuitive way and make them feel in control.

6.4 Request statistics

Integration of rate limiting comes with an additional extremely useful feature out of the box - request statistics in real time. We can use this for self-evident use cases and observability or some more advanced ones like load balancing. Load balancing increases availability, improves performance by increasing reliability, increases throughput, maintains, stability, optimises resource utilisation and provides fault tolerant capability [37]. Rate limiter holds all data needed in memory so we can efficiently query system's load and request statistics for a given sampling period to efficiently utilise load balancing. This way we can avoid expensive log aggregations that may otherwise be needed. This becomes even more handy in implementations with request weighting. Log aggregation queries would have to parse each request log for its weight to collect everything rate limiter has in memory.

Note on request statistics

It would be great to get detailed information about number of requests in a given sampling period per user, however statistics exposed by rate limiter are algorithm specific. Quality of exposed statistics differs and its interpretation should be adjusted to the implementation used.

Token bucket

Token bucket algorithm records only holds their token count. By subtracting token count from token capacity we can obtain number of requests issued (used tokens) in a given sampling period. Note that this is true only if all requests are weighted the same. Bucket is also being refilled periodically, so retrieving actual number of requests issued in a given sampling period is impossible. To illustrate the problem imagine a bucket with total capacity of 10 tokens and refill policy of 10 token per minute. At some point our API consumer issued 10 requests and used all of the tokens in the bucket. 15 seconds later he wants to know how many requests he did in last minute.

Rate limiter would answer with 8 while he actually did 10 requests in the last minute. Difference of 2 requests is due to the bucket being refilled consistently. 10 requests per minute means a new token is distributed every 6 seconds. In 15 seconds that would mean 2 new tokens in the bucket.

Fixed window

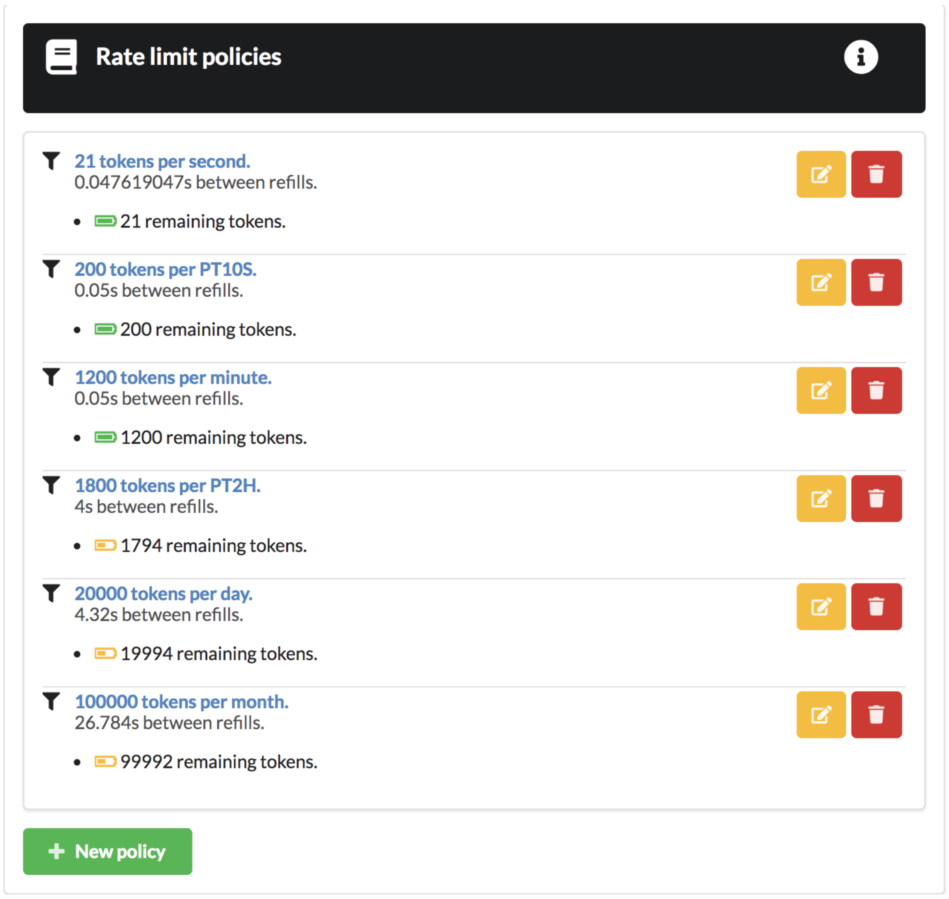
Fixed window suffers from slightly different problem. If user queries about data inside the same sampling period he did the requests in, everything is okay. As fast as the new sampling period start counters reset and all data from previous sampling period is lost. Imagine an 1 hour fixed window from 11:00-12:00. User might consume all of his tokens just before the new window start. Querying for number of requests in last hour after 12:00 would return 0 which might be confusing to the users.

Sliding window log

To obtain accurate request statistics we have to store all request timestamps. Sliding window log takes this approach but comes with its on drawbacks discussed in section 4.4.

Sliding window counters

Accuracy of request statistics in sliding window counters algorithm can be controlled through its parameter. With k=100 (dividing window into 100 slices), we miss only 1/100 of the requests (requests in bucket that just fell out of the sliding window slice) of a sampling period. Providing 99% request statistics accuracy, sliding window counters algorithm performs very well for load balancing purposes.


Rate limit policies

21 tokens per second.
0.047619047s between refills.
- 21 remaining tokens.

200 tokens per PT10S.
0.05s between refills.
- 200 remaining tokens.

1200 tokens per minute.
0.05s between refills.
- 1200 remaining tokens.

1800 tokens per PT2H.
4s between refills.
- 1794 remaining tokens.

20000 tokens per day.
4.32s between refills.
- 19994 remaining tokens.

100000 tokens per month.
26.784s between refills.
- 99992 remaining tokens.

Figure 6.2: Management web interface for intuitive rate limit contract manipulation in real time

Chapter 7

Evaluation results

We have been using rate limiting in production for quite some time. Over the next few sections we present results of its integration and effects it had on our core services (presented in the following section). Before proceeding with the implementation, we asked ourselves some questions to identify strengths, weaknesses, opportunities, and threats that rate limiting could bring us. Those are nicely summarised in the following SWOT matrix.

STRENGTHS	WEAKNESSES
1. Reduced average response times	1. Additional layer of complexity in the system
2. API Monetisation options, increased revenues	2. Maintain DBMS for storing rate limit records
3. Prevent service abuses	
4. Decreased infrastructure requirements and costs	

OPPORTUNITIES	THREATS
1. Attract more users by providing more reliable system	1. Lose existing customers by enforcing too strict rate limits 2. Bug in rate limiting implementation may crash the request (making service inaccessible as every request goes through the filter)

7.1 Setup

All of the services used for evaluation are deployed in the Amazon's cloud. AWS offers reliable, scalable, and inexpensive cloud computing services. To begin, sliding window counters algorithm discussed in section 4.5 was integrated into services implementing WMS standard. The WMS standard provides an interface for requesting geo-registered map images from one or more distributed geospatial databases. A WMS request defines the geographic layer(s) and area of interest to be processed. The response to the request is one or more geo-registered map images (returned as JPEG, PNG, etc.) that can be displayed in a browser application [38].

Scale

In our example, WMS services serve over 50 millions request per month with spikes of over 1000 requests per second. Those request may be extremely CPU intensive and render satellite images as seen on figure 7.1 up to 5000x5000 pixels at resolution of 10 meters. To handle such scale, we are using 8-12 compute optimized m5d.large instances behind a HAProxy load balancer.

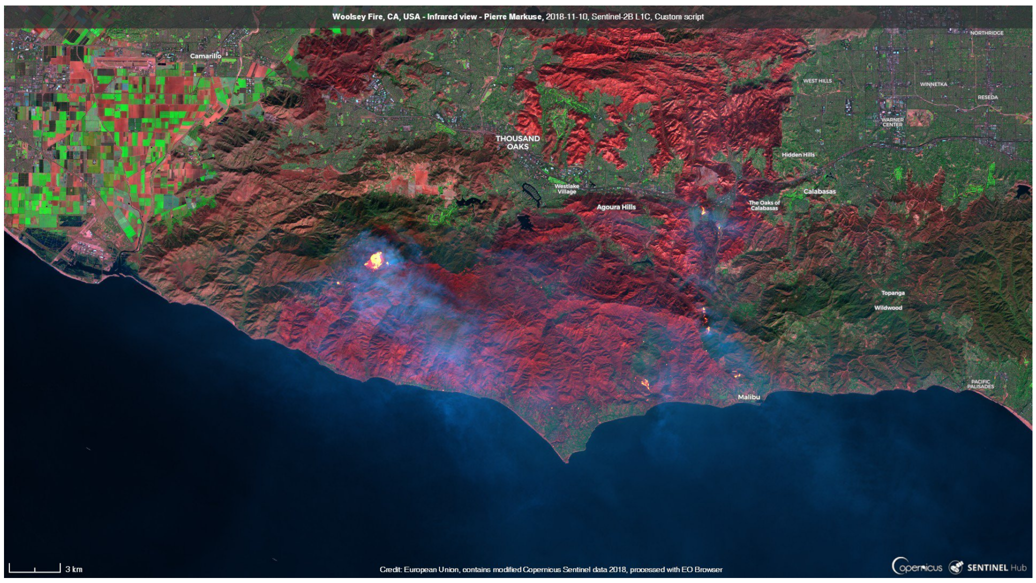

Woolsey Fire, CA, USA - Infrared view - Pierre Markuse, 2018-11-10, Sentinel-2B L1C, Custom script
Camarillo
THOUSAND OAKS
Westlake Village
Agoura Hills
Hidden Hills
Calabasas
The Oaks of Calabasas
WEST HILLS
WINNETKA
RESEDA
NORTHRIDGE
WARNER CENTER
Topanga
Wildwood
Malibu
PACIFIC PALISADES
3 km
Credit: European Union, contains modified Copernicus Sentinel data 2018, processed with EO Browser
SENTINEL Hub

Figure 7.1: Example response to a WMS request (multiple stitched WMS requests). Sentinel-2 Satellite data post processed with a custom fire script to highlight burn scars inflicted by California wildfires in November, 2018 [39].

API

All requests to our WMS API have to be authenticated and pass a rate limiting filter. Typical WMS requests as seen on figure 7.2 are very complex, searches over petabytes of satellite data and go through multiple phases.

1. Query Configuration service - business logic (fetch user configurations, check authentication,...)

2. Query Index service (get affected tiles by the request's area of interest - complex geometry intersection queries on PostgreSQL database)

3. Search AWS S3 for a JSON config with information about the affected tiles

4. Search AWS S3 for actual map images (JP2, JPEG, PNG,...)

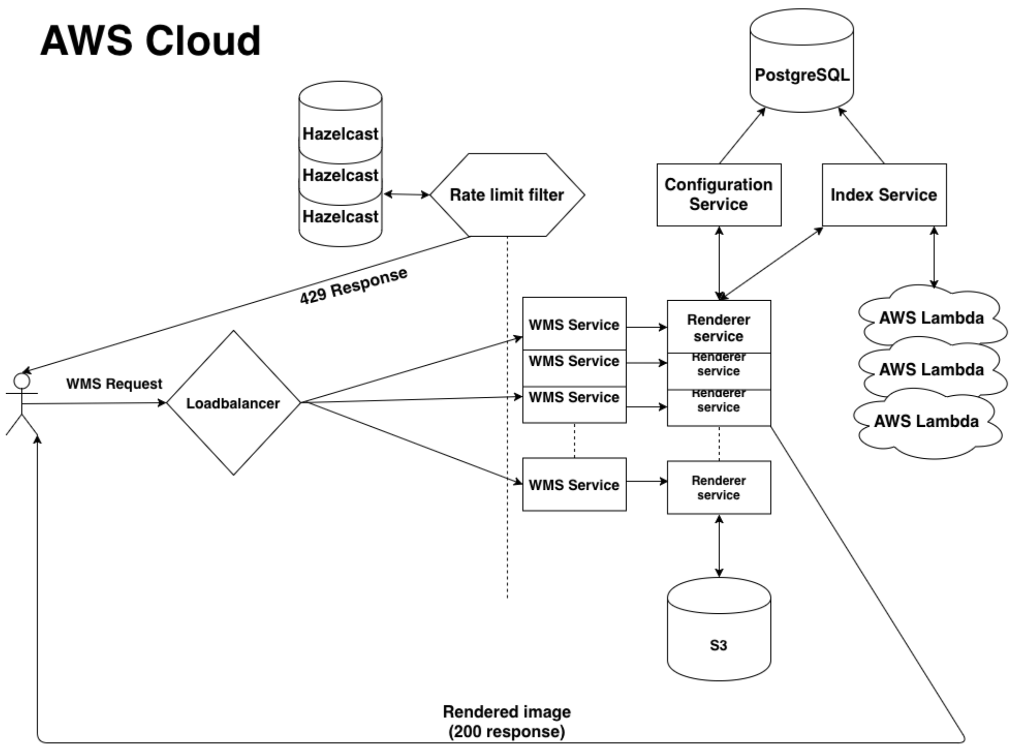


Figure 7.2: Lifecycle of a typical WMS request in a simplified setup arhitecture hosted on Amazon's cloud.

5. Execute user's custom script on the images using Javascript V8 Engine

6. Render image to the user

Weight of a WMS request may vary greatly, depending on its parameters (area of interest, time range, custom user script, satellite datasource,...). Calculating weight of such request upfront is very expensive due to complex geographic transformations, network request and database queries, thus approach proposed in chapter 3 was used. Limiting such request saved us a vast amount of CPU usage, network requests and database queries.

Storage

All rate limiting records are stored in a Hazelcast cluster. Hazelcast's replication technique enables Hazelcast clusters to offer high throughput, but suffers from best-effort consistency [40]. This is perfectly fine for rate limiting and should be negligible in most use cases. From our benchmarks, Hazelcast coupled with Java significantly outperformed other NoSQL key-value databases like Redis and Aerospike. Because Hazelcast is written purely in Java, it is able to perform significant optimisations through JVM. This heavily improved performance of serialization compared to other databases. Our Hazelcast cluster network consists of 3 nodes deployed on m5d.2xlarge instances providing high reliability, performance and throughput for our rate limiting filter. Moreover, we heavily utilised Hazelcast's Entry Processor which allows for code to be executed on the Hazelcast node itself (similarly to Lua scripting in Redis) to prevent race conditions. Before that, we used to fetch records from Hazelcast, do update logic (distribute new tokens, consume tokens) and store them back. Naturally, this led to race conditions, and an additional network request to the Hazelcast.

7.2 Response times

As rate limiting is done prior to every request being processed it is self evident that some response latency may be introduced. In our case, we achieved average rate limit execution times of 5.5-6ms while keeping 95th percentile consistently around 7 ms. This means we are able to rate limit 95% of all the requests in less than 7ms. This is a lot faster than we have expected and was possible due to the high throughput optimisations and configuration of our Hazelcast in-memory data grid [28]. Some outliers as seen on figure 7.3 were anticipated, but they occur infrequently and are usually due to our storage infrastructure cloud provider problems. We had no control over those, as they are unavoidable in distributed cloud based environments. To avoid significant overheads in such scenarios we have introduced a hard timeout

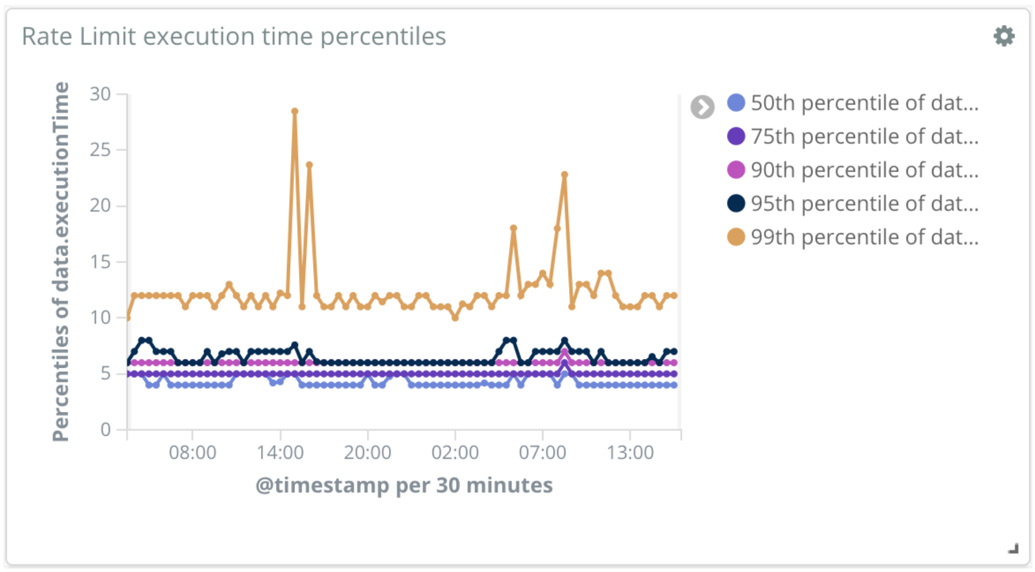


Figure 7.3: Rate limit execution time percentiles as seen from our monitoring tools. 95th percentiles is usually around 7ms. Outliers are result of our storage cloud provider infrastructure problems.

of 200ms to our rate limiting filter. If our storage cannot respond in this time, requests are simply passed through as if there was no rate rate limiting applied.

Prior to rate limiting integration, average response times from our core APIs resources were around 1.5 seconds. Applying rate limiting introduced around 0.4% overhead to our response times. We were completely okay with that as it is not noticeable to our users. However, it turned out that our average response times actually decreased since integration of rate limiting. We have observed reduced response times up to 20% in some scenarios. This usually occurred during high traffic bursts which previously overloaded our services. Figure 7.4 displays how such high traffic burst looks in action. It turned out rate limiting helped us reduce average response times, by reducing number of outliers, by around 8% to ~1.4s.

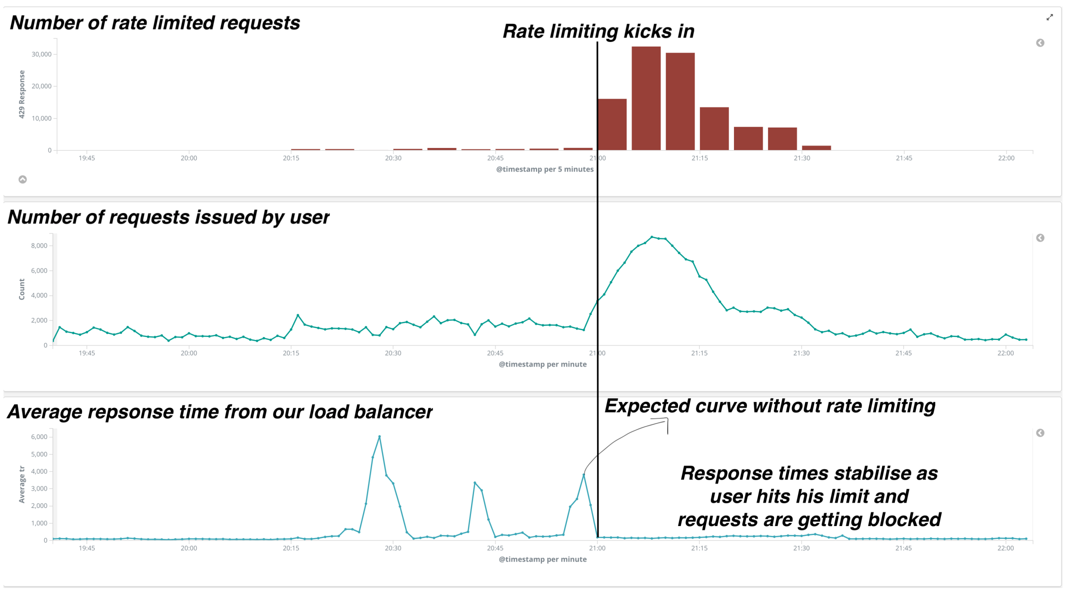


Figure 7.4: As rate limiting kicks in, average response times starts to stabilise as number of requests passed through the filter decreases.

7.3 Infrastructure

We always had a few extra backup instances running just in case of a big traffic spike. Only this way we were able to handle big traffic spikes and all of the requests normally. Rate limiting enabled us to get rid of those unnecessary servers. We are also heavily relying on serverless architecture and AWS Lambdas [41]. Rate limiting significantly reduced number of Lambda invocations and helped us reduce infrastructure requirements and billing by around 10%. We are now able to handle more (conformant) requests with less infrastructure. Moreover, utilising quick lookups to system's load from our rate limiting storage we were able to do load balancing much more accurately. This way we achieved much better usage of our infrastructure and instance provisioning which further improved our service stability and reliability.

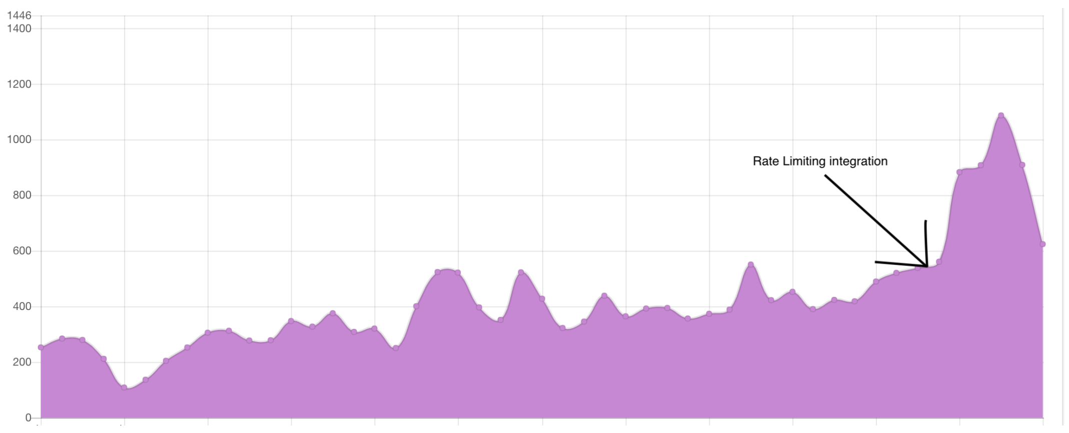


Figure 7.5: Number of users upgrading to paid subscription plans for higher rate limits. Y axis represents number of users, while X axis represents time - weekly data points.

7.4 API Monetisaion

Rate limiting enabled us to implement proper API monetisation. Enforcing rate limits increased our conversion rates of our paid accounts/subscriptions to obtain higher rate limit quotas. As a consequence, this nicely bumped our revenues from web APIs. We are now finally able to properly limit our "trial" users which are able to use our services free of charge for 1 month. As anyone could create a trial account, we observed some severe service abuses. To avoid such abuses, we enforced much stricter rate limits for our trial users. This prevented them from abusing trial accounts indefinitely and encourage them to upgrade and pay the subscription for higher request quotas. Spike in our paid subscription plans can be seen on figure 7.5.

Chapter 8

Conclusion

Rate limiting comes as one of the measures software engineers should integrate when facing scalability and reliability problems in web APIs. Moreover, it unlocks compelling opportunities to monetise web APIs. Integrating rate limiting into our system, we have significantly reduced number and magnitude of request spikes in our system. Suspicious increases in traffic are now under control, reducing our infrastructure requirements and costs to handle such traffic. Using approach proposed in chapter 3 we were able to accurately determine request weights without introducing additional overheads to response times. We have observed reduced average response times to up to 20% in some scenarios. Amount of errors and crashes was significantly reduced, as majority of those were occurring during request spikes we could not handle. Additionally, rate limiting storage was utilised for lookups about systems's load to perform load balancing. We were able to do that all of that without large memory footprint by implementing modified rate limiting algorithm proposed in section 4.5. Lastly, proper API monetisation and usage abuse prevention of our API was finally implemented. For future work, we would like to make rate limiting smarter by detecting abuses of individual licenses. This could further increase our revenues as we would force such users to upgrade to Enterprise plans for their teams. Such detection could be done by observing request IP access patterns, however it should be done very efficiently and in real time which poses a tough technical challenge.

Bibliography

[1] Mohammad Noormohammadpour and Cauligi S. Raghavendra. Datacenter traffic control: Understanding techniques and trade-offs. *CoRR*, abs/1712.03530, 2017.

[2] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. *SIGCOMM Comput. Commun. Rev.*, 37(4):337–348, August 2007.

[3] N.U. Ahmed, Qun Wang, and Luis Orozco-Barbosa. Systems approach to modeling the token bucket algorithm in computer networks. *Mathematical Problems in Engineering*, 8, 06 2002.

[4] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, and Changhoon Kim. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[5] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, second edition, 2006.

[6] Slavko Žitnik Aleš Kumer Marko Bajec Aljaž Zrnec, Lovro Šubelj. Podatkovne baze nosql. *Uporabna informatika*, 20(3):164–172, 2012.

[7] O. Bonaventure. *Computer Networking: Principles, Protocols, and Practice.* The Saylor Foundation, 2011.

[8] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, and Ying Zhang. Low latency software rate limiters for cloud networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, pages 78–84, New York, NY, USA, 2017. ACM.

[9] RFC6585 - Additional http status codes. `https://tools.ietf.org/html/rfc6585`. Accessed: 2018-09-18.

[10] What does a request mean? `https://sentinel-hub.com/faq/what-does-request-mean`. Accessed: 2018-09-25.

[11] Rate limiting Middleware in GO. `https://github.com/ulule/limiter`. Accessed: 2018-10-31.

[12] Rate limiting Middleware for Node.js. `https://github.com/jhurliman/node-rate-limiter`. Accessed: 2018-10-31.

[13] Kong. `https://konghq.com/kong-community-edition/`. Accessed: 2018-08-01.

[14] WSO2. `https://wso2.com/`. Accessed: 2018-08-01.

[15] Announcing Zuul: Edge Service in the Cloud. `https://medium.com/netflix-techblog/announcing-zuul-edge-service-in-the-cloud-ab3af5be08ee`. Accessed: 2018-08-27.

[16] Nginx API Gateway. `https://www.nginx.com/solutions/api-gateway/`. Accessed: 2018-08-27.

[17] API Umbrella - Open Source API Management. `https://apiumbrella.io/`. Accessed: 2018-08-27.

[18] Tyk Open Source API Gateway. `https://tyk.io/`. Accessed: 2018-08-27.

[19] J. Kidambi, D. Ghosal, and B. Mukherjee. Dynamic token bucket (DTB): a fair bandwidth allocation algorithm for high-speed networks. In *Proceedings Eight International Conference on Computer Communications and Networks (Cat. No.99EX370)*, pages 24–29, 1999.

[20] Han Seok KIM, Eun-Chan PARK, and Seo Weon HEO. A Token-Bucket Based Rate Control Algorithm with Maximum and Minimum Rate Constraints. *IEICE Transactions on Communications*, E91.B(5):1623–1626, 2008.

[21] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2010.

[22] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don'T Matter when You Can JUMP Them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 1–14, Berkeley, CA, USA, 2015. USENIX Association.

[23] Hongliang Li, Xuan Feng, Song Shi, Fang Zheng, and Xianghui Xie. A High-Accuracy Clock Synchronization Method in Distributed Real-Time System. In Weixia Xu, Liquan Xiao, Jinwen Li, Chengyi Zhang, and Zhenzhen Zhu, editors, *Computer Engineering and Technology*, pages 48–157, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[24] An alternative approach to rate limiting. `https://blog.figma.com/an-alternative-approach-to-rate-limiting-f8a06cf7c94c`. Accessed: 2018-07-27.

[25] Raymond Board. Distributed Database Systems. *IASSIST (International Association for Social Science Information Services and Technology) Quaterly*, pages 4–10, 1993.

[26] Matti Paksula. Persisting Objects in Redis Key-Value Database. Technical report, University of Helsinki, Department of Computer Science, 2018.

[27] Redis. `https://redis.io/`. Accessed: 2018-08-09.

[28] Hazelcast. `https://www.aerospike.com/lp/aerospike-community-edition/`. Accessed: 2018-08-09.

[29] Ben Evans. An Architect's View of Hazelcast. Technical report, Hazelcast Inc., 2015.

[30] Aerospike. `https://www.aerospike.com/`. Accessed: 2018-08-09.

[31] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a Real-time Operational DBMS. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[32] D.S.Hiremath and Dr.S.B.Kishor. Distributed Database Problem areas and Approaches. *IOSR Journal of Computer Engineering (IOSR-JCE)*, 2:15–18, 2016.

[33] L. England A. Glickenhouse. API Monetization – Understanding your Business Model Options. `https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=KUW12387USEN`, 2016. Accessed: 2018-08-09.

[34] Google Maps API pricing changes. `https://developers.google.com/maps/billing/important-updates#billing_changes`. Accessed: 2018-09-24.

[35] Rate Limiting at Github. `https://developer.github.com/v3/rate_limit/`. Accessed: 2018-09-18.

[36] Rate Limiting at Twitter. `https://developer.twitter.com/en/docs/basics/rate-limiting.html`. Accessed: 2018-09-18.

[37] Zahra Mohammed Elngomi and Khalid Khanfar. A Comparative Study of Load Balancing Algorithms: A Review Paper. *International Journal of Computer Science and Mobile Computing*, 5:448–458, 06 2016.

[38] What is an Open API? https://docs.hazelcast.org/docs/latest-development/manual/html/Consistency_and_Replication_Model.html. Accessed: 2018-11-12.

[39] Hot spots and burn scar from #MalibuFire #WoolseyFire #HillFire as seen by our #Sentinel2. https://twitter.com/CopernicusEU/status/1061544764334583808. Accessed: 2018-11-13.

[40] Hazelcast Consistency and Replication Model. https://docs.hazelcast.org/docs/latest-development/manual/html/Consistency_and_Replication_Model.html. Accessed: 2018-11-12.

[41] AWS Lambda. https://aws.amazon.com/lambda/. Accessed: 2018-11-12.