Technische Universität Berlin
Electrical Engineering and Computer Science
Department of Telecommunication Systems
Complex and Distributed IT Systems (CIT)

A Twitter Graph Capturing Framework

Bachelorarbeit

von Luk Burchard

zur Erlangung des Grades „Bachelor of Science“ (B. Sc.)
im Studiengang Computer Science (Informatik)

Erstgutachter:	Prof. Dr. habil. Odej Kao
Zweitgutachter:	Prof. Dr. Markus Brill
Erstbetreuer:	Daniel Thilo Schroeder
Zweitbetreuer:	Dr. Johannes Langguth

30 September

Erklärung der Urheberschaft

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Ort, Datum

Unterschrift

Zusammenfassung

Soziale Online-Netzwerke wie Facebook oder Twitter sind für die wachsende Bevölkerung ein wichtiger Ort, um Wissen und Informationen zu sammeln. Ein Trend zur Schaffung und Verbreitung schädlicher Fehlinformationen, oft als "fake news" bezeichnet, hat sich in den letzten Jahren entwickelt. Dieser wird von der WEF als Bedrohung für unsere moderne, stark verbundene [Web+16] Gesellschaft angesehen. Um entsprechende Phänomene zu untersuchen und zu verstehen, verwendet das UMOD-Projekt [Umo] beispielsweise einen graphentheoretischen Ansatz. Dabei werden Konversations– und Konnektivitätsgraphen aus Twitter extrahiert. Diese können dann genutzt werden, um einzigartige Merkmale zu identifizieren, welche für das Verbreiten von Fehlinformationen spezifisch sind. Allerdings stellt das effiziente Extrahieren großer Datenmengen von Twitter mit mehreren Tokens eine große technische Herausforderung dar. Diese Arbeit beschreibt das Design und die Implementierung eines Authentifizierungs-Proxys für die Twitter-API zur internen Verwendung. Der Proxy ist so konzipiert, dass die Datenquoten mehrerer Benutzer gleichzeitig verwaltet werden können und somit die Datenbeschränkungen von Twitter transparenter werden. Er skaliert linear in den verwendeten Clients und Token und verursacht dabei fast keine Leistungseinbußen. Es besteht damit auch nur geringer Implementierungsaufwand für weitere Layer bzw. Anwendungen, die mit der Twitter-API arbeiten müssen. Die Authentifizierung-Proxies, sofern sie mit genügend Token ausgestattet sind, machen damit eine hohe Skalierung bei niedrigen Leistungseinbußen möglich.

Abstract

Online social networks such as Facebook and Twitter are for a growing population an important place to gain knowledge and information from. A trend of creating and spreading harmful misinformation often called “fake news” has evolved in recent years and is by the WEF [Wef] considered to be a threat to our growing hyperconnected [Web+16] society. In order to investigate and understand this phenomenon the UMOD-project [Umo] uses a graph-theoretical approach. Here, the objective is to extract online conversations and connectivity graphs from Twitter with the emphasis to identify unique characteristics specific to this type of misinformation. However, scraping large amounts of data from Twitter using multiple tokens and endpoints efficiently poses a considerable technical challenge. This thesis describes the design and implementation of an authentication proxy to the Twitter API for internal usage. The proxy is designed in such a way that the data quotas of several users are managed simultaneously and thus Twitter’s data restrictions become transparent. The result is a proxy that scales linearly with clients and tokens in use and incurs almost no performance penalties or implementation overhead to further layer or applications that need to work with the Twitter API. The authentication proxies if provided with enough tokens can provide any scale at high performance with low penalties.

Contents

List of Figures

List of Tables

1 Introduction

Online social networks such as Twitter and Facebook have been rising in popularity. The advent and extreme user saturation of these new online social platforms allow individuals to participate in large hyperconnected [Web+16] networks. Like never before, a rising fraction of the population has the ability to share information and opinions in real-time as conversations are not bound by mediums that are limited by physical boundaries such as word-of-mouth.

Digital wildfires were observed and defined as "rapid spread of misinformation even when correct information follows quickly" [HWR13]. The World Economic Forum [Wef] warned of increasing misinformation disseminated in digital wildfires on social media platforms as a global threat to society. This phenomenon can be compared to a mass panic where information is quickly shared without verification. This misinformation can be created by accident or intentionally with a malicious goal in mind such as economic advantages or political agendas. For example the US election of 2016 and Brexit which have put "fake news" and "post-truth" into the global focus of scientists and the public.

There are not yet any well-established methods to automatically detect and control the spread of such misinformation[SPL19][Wan17]. Thus the analysis of misinformation's spread via online social networks has become crucially important, as such misinformation can have dramatic economic and personal impacts on a global scale. The UMOD [Umo] project tries to identify and analyze such misinformation. The project aims to develop tools and techniques to counteract misinformation spread.

For many experiments and studies UMOD requires a vast amount of complex and broad data from the Twitter network in a large scale which need to be available quickly in order to counteract on them. This poses a challenge as currently open sourced and pre-existing tools don't offer management and coordination of Twitter access quotas in a distributed way. For analysing the connections between people and information in those networks it is required to get an actual representation of the conversations and its history. Namely by extracting the spreading graphs of digital wildfires and then using machine learning to analyse the structure of the distribution patterns. The UMOD project builds upon the hypothesis that these distribution patterns are characteristic and thus will make it possible to detect wildfires through them. Other approaches try to find linguistic approaches such as sentiment analysis or falsification of messages with knowledge databases[CRC15].

Many networks like Facebook want to keep their user information closed to the general public to create a safer environment. Twitter, on the other hand, is used for scientific analysis, as user data is considered public. Due to this public nature it is used by politicians, celebrities, and companies. The public nature of Twitter makes possible to analyse mass data from the platform. Furthermore, it is also technically easier to acquire

the data as Twitter provides a well defined API to interact with the platform.

This thesis focuses on mitigating Twitter's API rate limit restrictions by implementing a proxy that transparently injects authentication information into proxied requests. The proposed approach allows continued use of existing tools and scraping infrastructure without modification, despite existing tools inability to handle distributed access to Twitter's API without triggering rate limit protections.

Twitter regulates access to its API with tokens that have an assigned quota (Section 2.1.2). By leveraging the proxy, existing software will continue to work as expected while the proxy provides a centralized API endpoint managing all available quotas in real-time.

The FACT [SPL19] framework was created to capture and analyse massive amounts of Twitter data in multiple experiment-environments. Those experiments can vary from bursts to long running jobs. The architecture is microservice oriented following best practices. This allows for a flexible setup with continuous modification during runtime and for addition or removal of multiple experiments during runtime.

2 Background

2.1 Twitter API

On average six thousand tweets a second are created on Twitter[Li+16]. To access this huge amount of data, Twitter offers a developer API that allows you to interact with the underlying services across multiple endpoints. Thus, everything that can be done on the website can be done by using the API. Yet Twitter does not allow to retrieve its entire data at once over their APIs. Therefore, enterprise customers have access to a so called Decahose[Twib] stream which includes 10% of all Twitter data in realtime. There is also a not listed option called "Firehose" which streams all Twitter data in realtime, this is not available to the public or research teams. There are some approaches to reconstruct these Firehose accesses by reverse engineering Twitters Snowflake IDs[Twid], but this is not the scope of this work and also violates Twitter's terms and conditions.

2.1.1 Twitter Quotas

Despite Twitters public and open user profiles and content, it does not allow for mass exporting the data, except for buying it through its Enterprise API which has higher limits then the public one. For normal purposes like 3rd-party Twitter apps, users have the ability to request an API token that allows the app to act and view content on their behalf. The REST-API is rate limited under a 15 minute time windows that resets the contingents of the individual endpoints to a constant amount of. Every URL path group has its own rate limit which we will call quota from now on. A URL path group is a matching regex path such as `/user/:id` where `:id` is a matching variable. In case that the quota is exceeded the API returns an HTTP "429 Too Many Requests". A time frame is anchored to the time of the first request on a given path. Furthermore, a User generated token has no global upper usage limit or rate limit meaning that we can continuously use the quota on an isolated path for analysis and calculations.

2.1.2 Twitter Authentication

Twitter requires requests to the API to be authenticated by an authentication token that is created by a user for an so-called Twitter app[Twia]. Here, Twitter uses OAuth 1.0a[Twic] to authenticate apps acting as a user on behalf of a user. OAuth is an authentication delegation protocol that generates tokens which carry the ability to authenticate as a target without knowing any sensitive user data such as the password. Twitter requires this OAuth token as an HTTP header field in every request to the API.

```
HTTP/1.1 200 OK
...
X-Rate-Limit-Limit: 75
X-Rate-Limit-Remaining: 21
X-Rate-Limit-Reset: 1561277231
X-Response-Time: 148
...
```

Figure 2.1: Example response Header

To increase the total amount of Twitter data that can be collected, the UMOD project uses a crowd-based approach. For this purpose, there is a donation button on the project website. If the button is clicked by a Twitter user, this user transfers his data contingent to the UMOD app in the form of an OAuth token. Currently, more than 90 users support the project. This means that the amount of data that can be collected is ninety times higher.

2.1.3 Twitter API Characteristics

Every Twitter REST-API response returns information about the current path rate limit status and other instrumentation related fields. These include the maximum possible usage amount on the current path, currently remaining requests and the epoch timestamp at which the used token quota gets reset.

2.1.4 Twitter Worst Case Analysis

Twitter is one of the main online social networks next to Facebook. Twitter started in 2005 with 5,000 tweets a day and increased by magnitudes to 35 million tweets per day in 2010 to finally almost 500,000,000 tweets per day in 2013[Li+16]. Twitter released these numbers in 2014, it can be assumed that these numbers haven't changed much due to the fact that Twitter's active user count has not increased after 2014. 500 million tweets a day equals 6000 tweets a second rounded up to the thousands on average. However during record tweet events like TV airings a twenty-fold increase was recorded with a peak of over 140 thousand tweets per second.

These numbers will be used to evaluate the feasibility and performance of the architectural choices and the implementation and design.

2.2 Golang

The proxy is written in Golang[Gol]. Go, short for the name Golang, is a statically typed, compiled programming language. It was designed and first implemented at Google to

replace their internal languages. Go has the CSP style concurrency natively built into the language and offers garbage collection as well as memory safety[Gol].

2.3 Containers

A container is a wrapper around different isolation layers in the kernel to run applications on multi user setups with replicable workflows. Containers bundles their whole environment such as (system) libraries, runtimes i.e JVM, scripts and executables. This makes containers self contained and allows them to be run on any system with the promise of working in exactly the same way. This also includes the production environment where the containers will be deployed to. The development can be done on a local machine with containers which saves development time and offers additional security over the written code and its side effects in a different environment.

2.4 Kubernetes

Kubernetes[Bur+16] is the container orchestration tool used to deploy the test environment, it is responsible for managing the life cycle of containers and to manage its surroundings such as the network, volumes and hardware resources. All components, e.g. the proxy or the broadcaster, which are developed in the scope of this work are deployed in a Kubernetes cluster to abstract away the complexity of maintaining a distributed system. Kubernetes offers first class functions to scale an application and make it publicly available. Additionally, it is possible to specify the structure of the cluster in manifests which are structured text files and push it to Kubernetes to reach this defined point automatically in a declarative manner.

2.4.1 Service

A Kubernetes Service is responsible for routing traffic to the proxy servers. This creates a load balancer which in a round robin manner distributes the connection to the set of running and healthy proxies.

3 Architecture

The general reason behind using a service to inject authentication information into a connection to Twitter is that we don't need an ongoing connection state or a centralized point to do so. This enables us to create a thin layer that is extensible with caching logic or metrics of how we interact with the Twitter API.

3.1 Architecture Overview

The architecture proposed here, is intended to extend the already existing framework architecture, proposed in the FACT paper[SPL19]. We already have a service which previously talked directly to the Twitter API. This scraper service also had to know of the OAuth tokens and was therefore responsible for managing quotas so that no congestion would occur. As Twitter authentication is done through OAuth we introduce an HTTP reverse proxy service that now is responsible for load balancing the tokens on the request to the real Twitter API.

3.1.1 Internal Reverse Proxy

A reverse proxy is an application that is executing queries, on behalf of a client, to another server. In our case the proxy address serves as the internal virtually unlimited twitter API given that we have enough user OAuth tokens and adequate load balancing. The proxy can be seen as a reverse proxy for the clients that need to be balanced onto the distinct OAuth endpoints. A single OAuth token can be seen as a different server in the backend. This design also allows for caching to further improve lookup times and reduce quota pressure.

As every proxy will be stateless, the proxy layer can be scaled horizontally to counter memory, CPU or IO stress. It is possible to scale the number of proxies given a scheduling system like Kubernetes (see Section 2.4) according to the demand.

This makes it possible to keep the current FACT-implementation as well as to scale and share the authentication layer so that the scrapers don't need to bundle connections on a centralised location. This approach also enables better distribution of token quotas between all scrapers and therefore it is no longer required to distribute the tokens to scrapers.

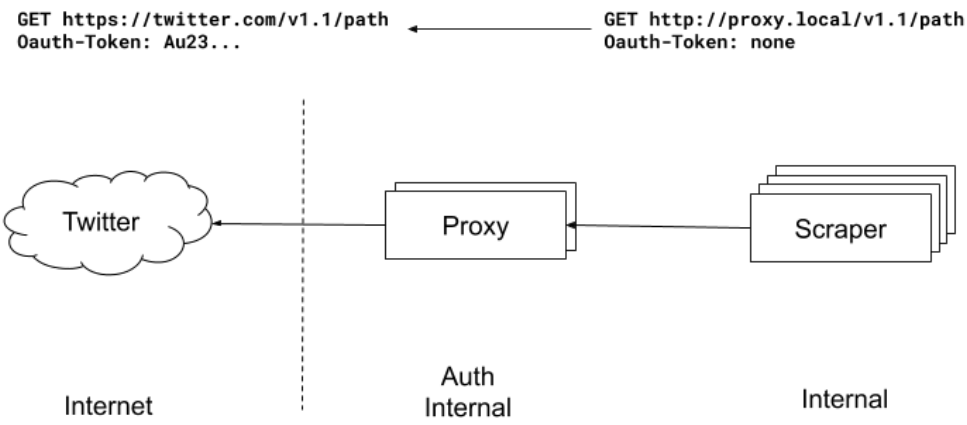


Figure 3.1: Layout of the front facing infrastructure

3.2 Token/Quota Distribution Algorithm

In order to implement the proxy aimed at in this thesis, tokens (Section 3.2) and their associated request quota are managed in such a way that as many requests as possible are successfully executed. The token distribution algorithm is the integral component to make the proxy scaleable and efficient over the use of multiple instances. Each proxy has access to all available tokens which can be considered as the maximum capacity of available requests. When increasing the number of processes that make use of quotas (Section 2.1.1) the proxy needs to coordinate each token's overall quotas. Moreover, the coordination has to be in a way that the proxy doesn't run into congestion problems where another proxy already exceeded the token quota. Here, congestion problems result in long round trip times, as the proxy has to retry the request with another token. There are two different ways of coordinating quota budgets with almost orthogonal properties. We can ether block every request and have a consensus algorithm in order to determine which proxy is going to use a token or we can work on a best effort basis to avoid global locks. These two methods can be categorized as blocking and non-blocking approaches (see Section 3.2.1 and 3.2.2). In the following section we will evaluate which proposed way is more suitable to implement.

3.2.1 Blocking

Blocking algorithms have in common that the globally remaining quota gets checked before executing a request. This prevents the sending of requests with exceeded quotas to the Twitter servers. The common problem with these algorithms is that a global lock needs to be acquired for performing a single request. This is time-consuming and computationally intense. Furthermore, all services are heavily dependent on the scheduler a locking service. Thus, in case of a critical service breakdown the the entire system

might be unable to operate and beyond that, it means that all services are dependent on a scheduler, which is responsible for managing the locks.

It would be possible to bypass this bottleneck and later optimize performance by sharding the quota groups but one pays for it with a not inconsiderable amount of additional complexity, maintenance and operational overhead.

3.2.2 Non-blocking

Compared to blocking algorithms non-blocking algorithms try to work on an greedy-best effort basis. This class of algorithms doesn't require global locking, information sharing and coordination to work. However algorithms in this class can still benefit from additional information that can be shared between the different clients. The additional information will improve the greedy algorithms to bring down the average case. Such extra information can be rate limits, other proxy server statistics, or request rates.

Non-blocking algorithms avoid the global locks by greedily trying to use locally remaining quotas with no perfect knowledge of the real remaining quotas which another proxy could have used. Moreover, non-blocking algorithms ideally use the extra rate limit information returned with every response from the Twitter REST-API to update their internal quota allowance list. Once a quota is exceeded with the last request or on a quota exceeded response the proxy marks the the quota as "not use". The quota will get reset after its rate limit lifetime passed.

3.2.3 Worst Case Analysis

The advantage of the blocking algorithm is to have *perfect* token usage without a single faulty request. It can be compared to creating a lease of a token budget and handing it out leading to an immediate success for every request. When working with a non-blocking greedy algorithm we don't get perfect token usage and in the worst cases request can fail, which will lead to another request until we (1) run out of tokens (2) have a successful request. This makes the worst case on a single request response time $\mathcal{O}(latency\_to\_twitter * \#tokens\_available)$. This approach will also lead to $\mathcal{O}(\#proxy * \#quota\_paths * \#quota_limit * \#tokens)$ faulty requests during each time-frame to the Twitter API. Later in Section 4 we evaluate more performance optimizations to drive down the faulty requests and the average request time.

3.2.4 Average Case

The average case of the blocking algorithm depends how long it takes to acquire a lock from the database and of the current load on the database. As with the non-blocking algorithms most requests are going to succeed without any further processing we can also set the average case to a low constant factor, as without locking a token can just be chosen.

3.2.5 Faulty Requests

Twitter does not document what the probable actions and consequences on too many requests on a rate limited endpoint are. Unauthorized quick accessing of an endpoint with a high request rate could be interpreted as a DDOS attempt. Twitter has a fair use policy in their terms of use and keeps the right to revoke tokens if they are not used responsibly or in an inappropriate manner. As this can cause problems for all experiments we try to minimize the faulty requests but don't require to keep the faulty request amount at zero. We will later set an appropriate faulty request rate target.

3.2.6 Security

Both approaches can be considered equivalent in their aspect of security. The most important security aspect to us is the number of tokens on a particular proxy instance which can be compromised. In case a proxy gets compromised the tokens that the proxy holds are leaking. We need to share all the available plain text tokens and thus all security relevant information as it gets injected into the HTTP request in plain text.

In the blocking approach the database which hands out leases to a quota also needs to send the OAuth token to the proxy. Therefore, it is possible that in a short time a single proxy is exposed to every token by going through all the global quota. Thus making all tokens visible to all proxies immediately should not make a difference as both approaches get exposed to all tokens over a short period. If there will be the need of a separation of tokens that is security relevant it is possible to run different clusters that are separated.

3.2.7 Discussion

Blocking algorithms have the advantage of perfect token usage but they require a complex setup and have high average call times. Moreover, in case of using a database it needs to be capable of withstanding peak burst workloads.

Since a Non-Blocking algorithm can be improved by introducing broadcasting events to other proxies that indicate when a quota was exceeded (Section 4.2) the latency as well as the number of faulty requests can be further reduced. Under high load, we can still run into the problem of checking every token multiple times and maybe come closer to the worst case but the majority of request will still run in low time. Not requiring a central component allows us to scale the proxies horizontally.

For the above mentioned reasons we have chosen the non-blocking greedy algorithm and optimize it to perform well and reduce the occurrences of the worst case making it possible to only consider the average case time complexity while reducing the occurrence of the worst case (Section 2.1.4).

3.3 Reverse Proxy Internals

The reverse proxy needs to be able to do mainly two things: rewrite and redirect requests with valid OAuth tokens and block requests with a retry time in case the corresponding

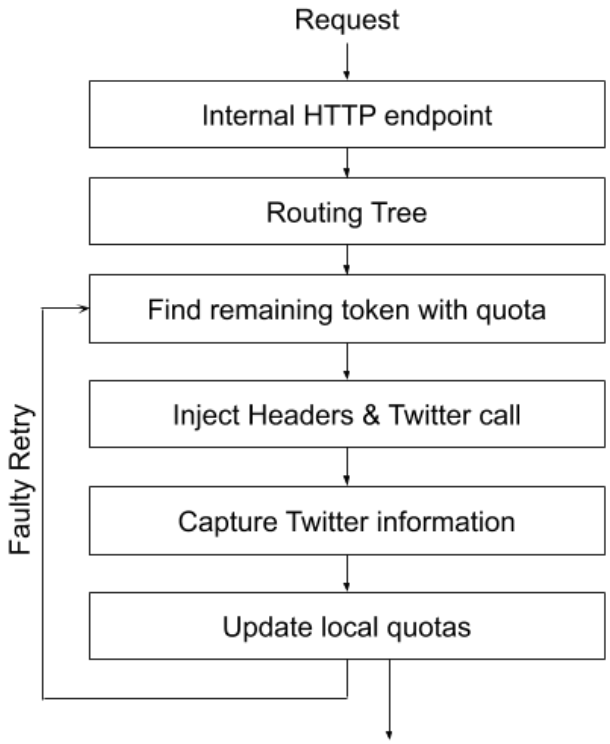


Figure 3.2: Layers of the internal proxy data flow: (1) The requested path gets parsed and matched (2) against its path group. (3) A token with remaining quota gets searched (4) and injected to the Twitter request. (5) The response gets parsed and local quota (6) adjusted. If the quota is already exceeded (7) another token gets chosen.

quotas are exceeded. The proxies job is to accept connections and to find an appropriate token to use.

3.3.1 Routing Tree

To quickly detect if a request has quotas available the corresponding URL path needs to be parsed and matched against its registered API URL-group which includes the rate limit information. Therefore, we need an efficient routing tree structure. The routing tree's **match()** function will be invoked for every request to get the meta path for a given requested path e.g. the real request path `/user/123` will match against the meta path `/user/:id`. We call those meta paths slugs, as they represent and contain the quota for a group of requests.

In our routing tree data structure, every full path points to the token-id and a quota tuple of $(available\_requests, reset\_tokens, reset\_epoch)$. Since our application should be able to parse the paths quickly and and all paths are static for an API version - only

```
...
"resources": {
  "users": {
    "/users/profile_banner": {
      "limit": 180,
      "remaining": 180,
      "reset": 1403602426
    },
    "/users/show/:id": {
      "limit": 180,
      "remaining": 180,
      "reset": 1403602426
    },
...
```

Figure 3.3: Example part of the `/application/rate_limit_status` endpoint

gradually changing over time when layout changes to the Twitter API happen. It seems to be a good practise to store all paths in a radix-tree with $log(250) \approx 8$ operations lookup time.

3.3.2 Constructing the routing Radix-Tree

A proxy at start time has no information about the Twitter API structure. We assume that the API structure is the same for every token. Twitter gives us the possibility to discover the structure of its endpoints. Under the path `https://api.twitter.com/1.1/application/rate_limit_status.json` we get all endpoints with their current quota and regex matching paths. It is necessary to get all paths in their regex notation as we can't simply group resource specific paths under one endpoint group.

We can use the request to dynamically construct the routing tree with all the available quotas. This would cause a startup dependency on Twitter. To avoid this dependency we can store the API responses during the build phase and use these during start time. The proxy later can execute requests to the Twitter API to keep its routing tree up to date.

3.3.3 Quotas Structure

As we need to construct our routing tree we need all slugs (Section 3.3.1) upfront. Ideally we can gather this information during the start-time. This can fail quickly due to two reasons (1) request fails (2) the quotas are already exceeded. Since this will make the startup time for the proxy slow we can create a service that offers this structure but having the same problems.

We inject an up to date structure during the build process so that the proxy can construct its routing tree immediately and the start-time is not increased.

Later it is possible to update the tree structure by executing requests to the rate limit API which returns the whole Twitter API structure.

3.3.4 Find remaining quota

After matching the correct slug to the group we need to find a token with remanining quota for the given time-frame. All tokens are stored in a list, we always choose a random starting point and from there on do linear check to all tokens if some quota still remains. Having found a token we return it to the next layer to work with it.

We will later discuss (Section 4.2) different ways of selecting tokens with quota to improve on the problem of collisions with multiple proxies.

3.3.5 Inject Headers & Twitter call

We overwrite the pre-existing OAuth headers and then call the Twitter endpoint. For performance reasons we maintain a set of open connections to the API. Without these open connections a TLS and TCP handshake would have to be established for every API call and therefore would be slower.

3.3.6 Capture Twitter information & Update local quotas

As described in Section 2.1.3 the Twitter API returns valuable information in its HTTP response headers. The local representation of the token quotas are getting adjusted to the state the Twitter API responded.

3.4 Implementation

3.4.1 Programming Language

For the implementation of the proxy server we require a language that supports good abstraction over concurrency and offers strong types for the security of the internet facing system. For this purpose we have chosen Go (Section 2.2).

3.4.2 Previous Codebase

The previous codebase already had a scraper and an internal rate limiter in a single java application that already utilised concurrent patterns to reduce the time while waiting for a network request/response. The scraper code also got basic login for rate limit quota avoidance and handling of multiple tokens. The new code is located under the /proxy folder and contains all code related to the project. The old and still used Java scraper code got stripped of it's unused code and adjust the Twitter base URLS to the proxy URL.

3.4.3 Load Balancing

The proxy is horizontally scaleable by adding multiple instances to a pool of proxies. We need to load balance the connections to the proxies by adding load balancers in front of those. The proxy does not require any state such as session affinity thus it is possible to use stateless round robin without concerns.

3.4.4 Problems

During testing the scraper extensively works with the rate limit endpoint and with rate limit information from the response header. The scraper uses this rate limit information to throttle the request amount to avoid blocked requests to the API server. When simply forwarding the request the local per token quota gets send. This makes the scraper assume that the quota is globally exceeded because it doesn't know of the individual quotas of all other tokens.

To avoid that scrapers stop before the global quota is exceeded the proxy has to track the global remaining quota during all requests. When proxying a request from the API the headers get exchanged to the global quota remaining limits and reset time.

As scrapers make extensive use of the **/application/rate_limit.json** endpoint it is required to replicate the behaviour of the endpoint. The endpoint returns the global quota of all tokens together so that the scraper knows of the global limits.

4 Experiments

To simulate how the proxy behaves in a real world scenario we are conducting experiments which have the goal to validate or fix the performance of the proxy. We are using up to 1600 simulated tokens in the proxies to simulate a real world scenario which would be able to scrape the real Twitter API in real time (Section 2.1.4). The UMOD project does not have access to the required 1600 tokens this is why we introduce a caching service which is shimming (Section 4.0.2) and reimplementing behaviour of the real Twitter API. This way we can have fake 1600 tokens, which act like real tokens to the shim. The shim is passing the first request to Twitter and is then using the cached result to respond to the proxy. The clients for benchmarking are requesting different Twitter user profiles, timelines and messages as in a real world scenario.

4.0.1 Load Generation

To generate load a tool called Apache Bench short AB[Abw] is used. AB generates parallel requests to an endpoint and plots the responses for latency and response codes. AB is used to test the API.

4.0.2 Shim

To reliably test against the Twitter API with controllable defined side effects and behaviour another proxy was built. This proxy is called shim because it shims the Twitter API to the quota load balancing proxy. The shim caches requests for an endpoint group and never invalidates the cache. The shim reimplements some Twitter API functionality, such as quota saving and decreasing. Arbitrary quota limit manipulation, latency injection and faulty responses and request timeouts to reproduce real world scenarios. With Apache Bench (Section 4.0.1) the shim can serve roughly 11k requests in a second with a payload of 70KB generating a throughput of 700-800MB/s.

4.0.3 Instrumentation & Metrics

To understand how the proxy performs, we need to collect program level metrics such as response code counts and request rates. For collecting and aggregating metrics from multiple applications we use the Prometheus framework. "Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud."[Pro] We choose Prometheus as it is easy to setup with Kubernetes and our Go code base.

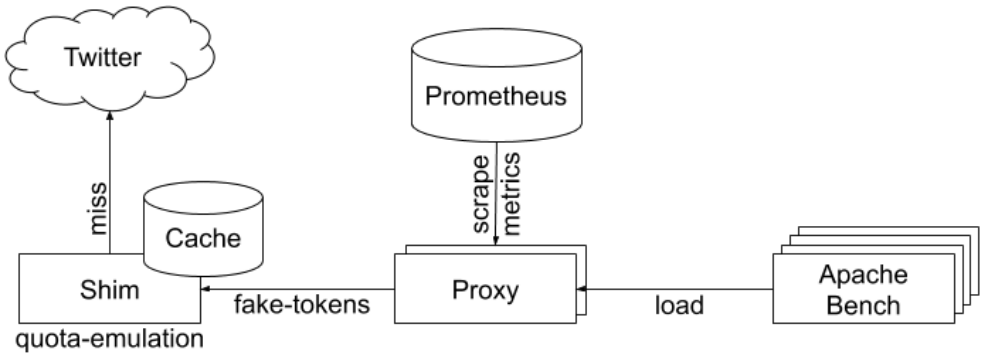


Figure 4.1: Layout of the test setup. (1) Apache Bench creates load to the proxy, (2) the Proxy is connecting to the shim with fake tokens. (3) The shim is caching responses for API groups, on a cache miss the real (4) Twitter API is called. (5) Prometheus scrapes the proxy for metrics

4.1 Throughput optimization

4.1.1 Setup

To understand the performance of more complex setup it important to understand the performance of a single instance. To get the most throughput the shim and the proxy that is being tested are run on the same machine. This allows for more IO which is done over localhost as the packets don't have to pass the whole TCP stack and are directly copied to the other programs memory.

To measure the bottlenecks of the proxy a profiling tool is injected into the program during compile time. The profiling library collects probes and metrics which later can be analysed by a tool such as pprof. Pprof is a visualization tool which can display Linux perf[Wea] files for mutex, CPU and memory profiles. Using pprof it is possible to visualize and find bottlenecks.

We do not introduce latency or not enough token quota so it's possible to get the maximum throughput. The performance will decrease when the applications get distributed on multiple machines with a network in between them.

4.1.2 Optimization one

When using the described setup utilizing mutex profiles it was notable that a decent amount of time was spend in the scheduler to set global quotas and printing log statements. Moving the mutex up to avoid waiting for log IO allowed for an increase in performance.

4.1.3 Optimization two

Using the CPU and mutex profiler another hot-spot after section 4.1.2 was visible when printing log statements which were sent when a limit was adjusted or a request forwarded.

Eliminating unnecessary log statements allowed for another increase in performance.

100 connections with 1600 tokens	
Experiment Name	Requests/s
Base	5404
Mutexs	7187
Minimum Log	11466

4.2 Pre-Advertising Requests (Broker)

4.2.1 Setup

This is an optimization to reduce the time and faulty-request average case. The current implementation does not allow for multiple proxies to share the state of the local token quotas, this leads to faulty requests to the Twitter API and long round trip times as a request needs to go through every token until the proxy locally knows that the global quota is already exceeded. To avoid running in already globally empty tokens a broadcaster is used to distribute information in a Pub/Sub[] manner. The proxies advertise when the quota of a token is locally exceeded so that all other proxies will not run into the worst case scenario.

Using a message broker does not increase the delay as information can be send asynchronously.

Each proxy subscribes to the broadcaster and applies events to it's quota resource pool. The messages from the broadcaster can only reduce the remaining limits, as information can be outdated when reaching another proxy. This way a minimum number of remaining token quotas is always being held by all proxies.

The test uses a single shim which is representing the Twitter API. As the shim runs in the local cluster the response time is artificially increased to around 100ms, normal distributed with a desired standard deviation of 25ms and a desired mean of 100ms $delay \sim \mathcal{N}(100, 25^2) + internal\_delay$.

The proxies use 100 tokens and have 600 asynchronous client connections to the pool of proxies which generate load.

The goal is to minimize the total amount of faulty requests

Broadcasting an exceeded token can be helpful. Still the problem of many requests at the same time does exist, though this is unlikely to happen on all tokens as the tokens get randomly selected.

All the information that gets advertised through the broadcaster is a representation of the global state, all proxies hold a local variation of this state.

When a new proxy joins the network it's representation of the token quotas has a high difference in token quota budget to the globally advertised budget. While the proxy subscribes to the global event stream it takes a whole Twitter rate limit time frame to get all information for one frame. During the 15 minute time frame until capturing

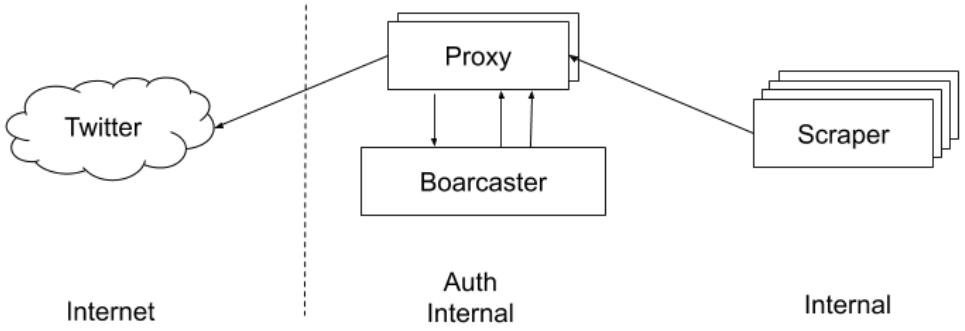


Figure 4.2: Integration of a message broker for lower bound correction

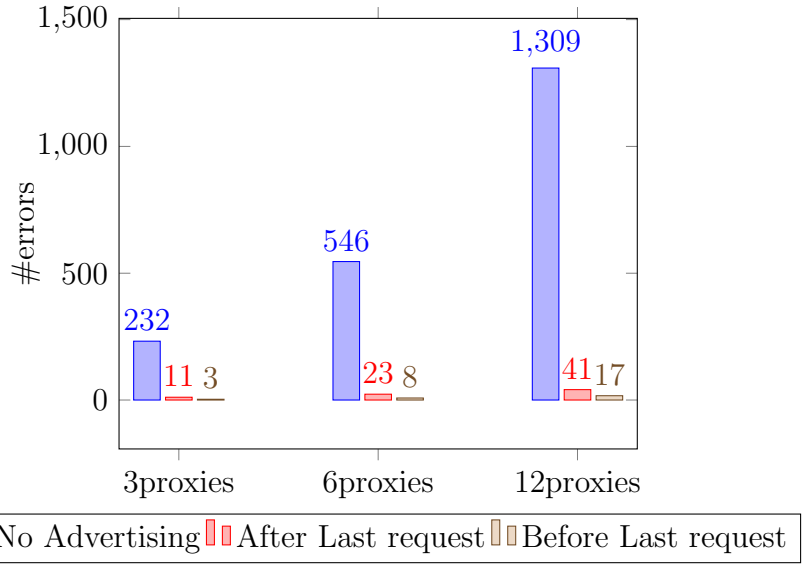
all events the proxy can run into worst case scenarios by accessing tokens that have no quota left. As every information gets reset over the course of 15 minutes, there is no need for replays of pub-sub messages except for optimization.

4.2.2 Implementation

The broker can be any kind of Pub/Sub system as long as it can distribute messages to all connected proxies in a performant manner. For message distribution Redis, a KV store with PUB/SUB primitives was chosen.

During testing Redis could offer 80k msg/s. In larger scenarios this can be a bottleneck. Using a software such as NATS which offers 3x the throughput might be suitable in later scenarios.

4.2.3 Evaluation



When running with one proxy the local state is always perfectly matching the global state. When adding more proxies each proxies local state differs from the global state. It's visible that when adding a proxy each proxy besides the first proxy that reaches a limit runs into the rate limit. This is why the sum of faulty requests is based on the number of proxies. When generating asynchronous load multiple requests can be forwarded by a proxy before adjusting to the global state of the token. This is why the numbers of faulty requests is higher then $(\#proxies * \#tokens)$.

When advertising the quota other proxies can adjust their local state.

5 Summary

5.1 Result

We can conclude that it is possible to build a transparent authentication layer which load balances request in a way that no central component will be needed. Furthermore, the proxy used to authenticate requests can be constructed in such a way that no consensus algorithm will be needed, thus avoiding a global lock. This novel approach can be used to support at least 1600 tokens over multiple proxies which is enough to recreate a full Twitter live stream. The code change to use the proxy is very limited as only a single address needs to be changed, by code or by DNS.

5.2 Outlook

It is possible to further improve the code, as for example a global state approximation is missing that's required for new proxies joining a network of preexisting proxies. In the current code a new proxy automatically assumes all tokens full before hitting the API endpoints. A global state approximation can help new proxies recover to a state that is generally accepted by everyone. It can also help to deal with missing messages from the broker. It is currently not implemented as experiments are long running and a proxy will recover to an optimum in the 15min time frame of the Twitter API.

Literature

[Abw] *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4*. URL: https://httpd.apache.org/docs/2.4/programs/ab.html (visited on 09/28/2019) (cit. on p. 23).

[Bur+16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. „Borg, omega, and kubernetes“. In: (2016) (cit. on p. 13).

[CRC15] N. J. Conroy, V. L. Rubin, and Y. Chen. „Automatic deception detection: Methods for finding fake news“. In: *Proceedings of the Association for Information Science and Technology* 52.1 (2015), pp. 1–4. URL: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/pra2.2015.145052010082 (cit. on p. 9).

[Gol] *The Go Programming Language*. URL: https://golang.org/ (visited on 09/28/2019) (cit. on pp. 12, 13).

[HWR13] W. L. Howell, World Economic Forum, and Risk Response Network. *Global risks 2013*. en. OCLC: 835917006. Cologny/Geneva, Switzerland: World Economic Forum, 2013 (cit. on p. 9).

[Li+16] Q. Li, S. Shah, M. Thomas, K. Anderson, X. Liu, A. Nourbakhsh, and R. Fang. „How Much Data Do You Need? Twitter Decahose Data Analysis“. In: July 2016 (cit. on pp. 11, 12).

[Pro] *Overview | Prometheus*. URL: https://prometheus.io/docs/introduction/overview/ (visited on 09/28/2019) (cit. on p. 23).

[SPL19] D. T. Schroeder, K. Pogorelov, and J. Langguth. „FACT: a Framework for Analysis and Capture of Twitter Graphs“. In: *The Sixth IEEE International Conference on Social Networks Analysis, Management and Security(SNAMS-2019)*. 2019 (cit. on pp. 9, 10, 15).

[Twia] *Authentication and authorization*. en. URL: https://developer.twitter.com/en/docs/basics/authentication/overview/authentication-and-authorization.html (visited on 09/28/2019) (cit. on p. 11).

[Twib] *Decahose stream*. en. URL: https://developer.twitter.com/en/docs/tweets/sample-realtime/overview/decahose.html (visited on 09/28/2019) (cit. on p. 11).

[Twic] *OAuth with the Twitter API*. en. URL: https://developer.twitter.com/en/docs/basics/authentication/overview/oauth.html (visited on 09/28/2019) (cit. on p. 11).

[Twid] *Twitter IDs (snowflake) — Twitter Developers*. URL: https://developer.twitter.com/en/docs/basics/twitter-ids (visited on 09/28/2019) (cit. on p. 11).

[Umo] *UMOD: Understanding and Monitoring Digital Wildfires | Simula Research Laboratory*. en. URL: https://www.simula.no/research/projects/umod-understanding-and-monitoring-digital-wildfires (visited on 09/18/2019) (cit. on pp. 1, 2, 9).

[Wan17] W. Y. Wang. „"Liar, Liar Pants on Fire": A New Benchmark Dataset for Fake News Detection". In: *arXiv:1705.00648 [cs]* (May 2017). URL: http://arxiv.org/abs/1705.00648 (visited on 09/27/2019) (cit. on p. 9).

[Wea] V. Weaver. „Linux perf event Features and Overhead". en. In: (), p. 35 (cit. on p. 24).

[Web+16] H. Webb, M. Jirotka, B. C. Stahl, W. Housley, A. Edwards, M. Williams, R. Procter, O. Rana, and P. Burnap. „Digital Wildfires: Hyper-connectivity, Havoc and a Global Ethos to Govern Social Media". In: *SIGCAS Comput. Soc.* 45.3 (Jan. 2016), pp. 193–201. URL: http://doi.acm.org/10.1145/2874239.2874267 (visited on 08/20/2019) (cit. on pp. 1, 2, 9).

[Wef] *Digital Wildfires in a Hyperconnected World*. en-US. URL: http://wef.ch/GJCg5E (visited on 08/21/2019) (cit. on pp. 2, 9).