Lecture 15 — Rate Limits

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 11, 2024

It’s Not Me, It’s You

Speed Limit

Sometimes the limiting factor in an app is not under our control.

A rate limit is exactly what it sounds like.

More Complex

Rate limits can be more complicated than just requests per unit time.

Multiple thresholds (e.g., A requests per hour or B requests per day)?

Max C requests to change your user data per day, and a max of D requests of any type per day in total?

Slow down vs just reject? We'll just talk about rejection for now.

Rejection Hurts

Rejected requests can be a huge problem for your application or service.

Obvious example: using ChatGPT in your app?

Other example: validating webhooks that the service itself sent in...

Why Are Limits?

Rate limits exist because every request has a certain cost associated with it.

The cost may or may not be measured in currency.

Consider opportunity cost.

DoS and DDoS

Denial of Service (DoS) attack: negatively impact some service by submitting many invalid requests to overwhelm it.

If using numerous clients: Distributed Denial of Service (DDoS) attack.

Rate Limit Example

Using a system that allowed Ontarians to write letters to the Minister of the Environment about allowing rock climbing in provincial parks.

We were encouraging people at climbing gyms to send letters (well, emails).

The climbing gym IP address got rate limited because people were sending letters from the gym's wifi.

Request Consume Resources

Consider the Unity Engine's proposed (retracted) changes in 2023.

Charge per installation? What if I write an install bomb?

Scraping All Data

Rate limits can also prevent scraping all your data.

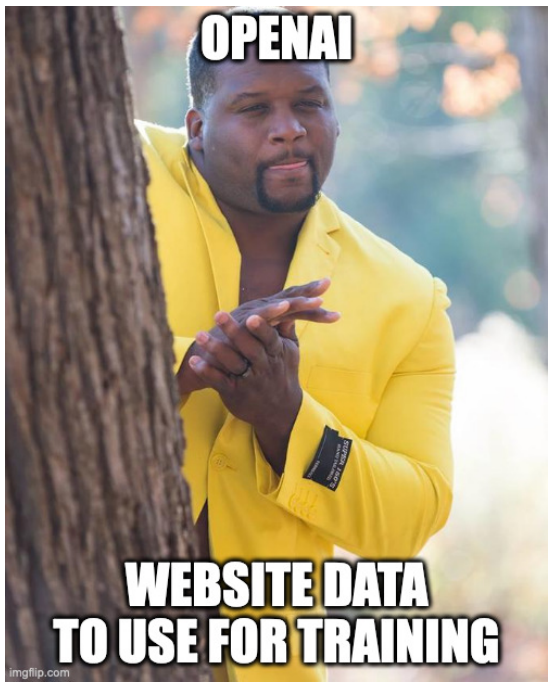
This really happened to the openly-political social media site “Parler”.

Two bad decisions: sequential post IDs and no rate limits.

For Whose Benefit?

Concern: if others can scrape all your content for their own benefit at your cost.

Even if it's intended as non-malicious training some machine learning model.



Yes, OpenAI did scan the internet to train ChatGPT.

Dealing with Rate Limits

Assume we have a system with a rate limit and we need to address that.

If the rate limit is super high compared to usage, no problem.

Hitting the rate limit regularly? Frustrating, or could get us banned!

What's the Limit?

If we get rejected requests we know there's a problem, but what's the limit?

- Does the documentation tell us?
- Is there an API to query it?
- Does it come in the responses?
- None of the above?

Why not tell us?

Atlassian Makes No Promises

“REST API rate limits are not published because the computation logic is evolving continuously to maximize reliability and performance for customers”

It does give them more freedom?

Testing for the limit here might work but information is out-of-date quickly.

What Can We Do?

Giving up: not the right answer... especially in an interview.

Can we do less work? Might help, but how much redundancy will we find?

Caching

Remember answers previously received.

Remember what we discussed about write-through and write-back caches.

It can be difficult to know if the data is changed remotely.

Use domain knowledge here! e.g., exchange rate quotes valid 20 minutes.

Group Up

Can we group up requests into a larger one?

There do need to be things to group with...

And the remote API has to support it.

Bigger Requests, Bigger Parsing

Grouping requests may also make it hard to handle if there's a problem with one of the elements in it.

If we asked to update five employees and one of the requests is invalid, is the whole group rejected or just the one employee update? (Is it a transaction?)

Consider the risks of more complex logic.

Patience

If the problem is that too many requests are happening in a short period of time, maybe our best solution is to distribute the requests over more time.

Patience

When there is a more demand for something than capacity, what do we do?

Simply controlling the rate at which requests leave the queue is sufficient to ensure that the rate limit is not exceeded.

Rust Crates

Unsurprisingly, multiple Rust crates do what we want.

Below are some examples from the documentation of the `ratelimit` crate.

This is not especially fancy, but is sufficient for an example.

Rust Rate Limit

```rust
use ratelimit::Ratelimiter;
use std::time::Duration;

// Constructs a ratelimiter that generates 1 tokens/s with no burst. This
// can be used to produce a steady rate of requests. The ratelimiter starts
// with no tokens available, which means across application restarts, we
// cannot exceed the configured ratelimit.
let ratelimiter = Ratelimiter::builder(1, Duration::from_secs(1))
    .build()
    .unwrap();

// Another use case might be admission control, where we start with some
// initial budget and replenish it periodically. In this example, our
// ratelimiter allows 1000 tokens/hour. For every hour long sliding window,
// no more than 1000 tokens can be acquired. But all tokens can be used in
// a single burst. Additional calls to `try_wait()` will return an error
// until the next token addition.
//
// This is popular approach with public API ratelimits.
let ratelimiter = Ratelimiter::builder(1000, Duration::from_secs(3600))
    .max_tokens(1000)
    .initial_available(1000)
    .build()
    .unwrap();
```

Rust Rate Limit

```rust
// For very high rates, we should avoid using too short of an interval due
// to limits of system clock resolution. Instead, it's better to allow some
// burst and add multiple tokens per interval. The resulting ratelimiter
// here generates 50 million tokens/s and allows no more than 50 tokens to
// be acquired in any 1 microsecond long window.
let ratelimiter = Ratelimiter::builder(50, Duration::from_micros(1))
    .max_tokens(50)
    .build()
    .unwrap();

// constructs a ratelimiter that generates 100 tokens/s with no burst
let ratelimiter = Ratelimiter::builder(1, Duration::from_millis(10))
    .build()
    .unwrap();

for _ in 0..10 {
    // a simple sleep-wait
    if let Err(sleep) = ratelimiter.try_wait() {
            std::thread::sleep(sleep);
            continue;
    }
    // do some ratelimited action here
}
```

Users Are Impatient

Enqueueing a request is not always suitable if the user is sitting at the screen and awaiting a response to their request.

It takes a synchronous flow and makes it asynchronous.

Rearchitecting it can be a long-term goal, or just something we do with requests that don’t have to be synchronous to make room.

Schedule Later

For requests that are not urgent, then another option is to schedule the requests for a not-busy time. (Conversely, CPUs have burst mode, which makes them go faster when busy).

Overnight might be a great time to do things that count against the limit.

Billing Examples

Imagine that you have a billing system where the monthly invoicing procedure uses the majority of the rate limit.

If that happens during the day, then there's no capacity for adding new customers, updating them, paying invoices, etc.

The solution then is to run the invoicing procedure overnight instead.

Billing Examples

Maybe you can even convince management to make it so not all users are billed on the first of the month, but that might require some charisma.

Which leads us to the next idea…

Roll Persuasion

Convince the other side to raise the limit?

Sometimes it’s negotiated, sometimes you need to pay more money, sometimes it’s just not possible.

It Happened Anyway

We might still encounter the occasional rate limit.

That’s not a disaster, as long as we handle it the right way.

The right way is not try harder or try more; if we are being rate limited, then we need to try again later. But how much later?

The Answer is Right There

Sometimes the answer for how long to wait is in the API response.

... if we read it.

Exponential Backoff

Resources might not be available right now; retry later?

It’s unhelpful to have a tight loop that simply retries as fast as possible.

Wait a little bit and try again; if the error occurs, next time wait a little longer.

Press F5 Until the Website Loads!

Repeatedly retrying doesn't help.

If it's down for maintenance, it could be quite a while before it's back.

Or, if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse!

Jitter

The exponential backoff with jitter strategy is good for a scenario where you have lots of independent clients accessing the same resource.

If you have one client, maybe you want something like TCP congestion control.