ECE459: Programming for Performance Winter 2024

Lecture 15 — Rate Limits

Jeff Zarnett *2023-10-16*

It's Not Me, It's You

Sometimes the limiting factor in our application is something that's not under our control. Whenever your application connects to some other application (whether external or not), we may run up against a rate limit.

A *rate limit* represents the maximum rate at which a requester can make requests of the service—it is exactly what it sounds like.

In other words, how many requests can be submitted in a given unit of time? Requests above the limit are rejected. If we are making HTTP requests, we are supposed to get a response code of 429 for a request rejected due to a rate limit.

Rate limits can be more complicated than just a simple measurement of requests per unit time. For example, there can be multiple thresholds (e.g., A requests per hour or B requests per day), and it may be segmented by request type/responses (e.g., max C requests to change your user data per day, and a max of D requests of any type per day in total). Responses can be more complicated than just rejection also—the service could intentionally delay or deprioritize requests that exceed a threshold—but for our discussion we'll just stick with simple rejection.

Rejected requests can be a huge problem for your application or service. An obvious example here is something like ChatGPT: if you're using that as a service in your application and you run up against the rate limit for that, some aspect of your application (maybe the critical one?) is unavailable or not working as well as expected. I [JZ] have even got a fun story about getting rate limited by a payment processing platform because they would (1) generate an invoice and notify a service via a webhook, (2) the service would then validate the webhook notification is valid by calling the platform, and (3) then get rate limited because the calling application is trying to call the validation endpoint too often in a short period of time. Wait, they're calling us and we're just checking that the call is valid—but unfortunately, they don't care, HTTP 429—Too Many Requests!

Why is this happening to me? Rate limits exist because every request has a certain cost associated with it. It takes work, however small or large this might be, to respond to the request. The cost may or may not be measured in a currency—if you are paying for CPU time at a cloud provider, then it literally is measured in monetary units—but it can also be measured in opportunity cost. What do I mean? If the system is busy, responding to one request may mean a delay in responding to other requests. If some of those requests are fraudulent or otherwise invalid, it's taking away time or resources from other, legitimate requests. If the system is overprovisioned, rate limits aren't necessary, but here we're talking about systems that are running closer to the edge.

Denial-of-Service (DoS) attacks are ways that attackers can negatively impact the functioning of a service by simply sending in many invalid requests. The regular DoS attack becomes a Distributed Denial of Service (DDoS) attack when the attacker sends in those requests from numerous clients. This not only allows the attacker much more capacity to send in requests, but also prevents simple solutions like blocking the IP of the offending system. Given enough invalid requests, it can overwhelm the service which will either cause it to crash, exhaust its resources (e.g., network handles), or be so slow as to be unusable.

An example of a rate limit that I [PL] encountered was in 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), but then the climbing gym IP address got rate limited because people were sending letters from the gym's wifi. We never did get a satisfactory resolution from the letter writing service. (I'd claim this is also an example of knowing the local context: we think that Canadians have lower data limits

than Americans, and thus more likely to be on public wifi rather than just sending from their own connections; the letter writing service didn't consider that local context.)

In another illustration of how requests can take up resources, we can also consider a proposed, and later retracted, approach that the Unity engine wanted to take in 2023 [Bat23]. Unity makes a game engine that people use to make fun games. The company's licensing model didn't really take into account the Live-Service kind of game (think Fortnite or Diablo 4 or similar) where the game is "free to play" but the company that makes it gets the money though selling cosmetic items, power boosts, or just generally preying on people with gambling addictions (via lootboxes which are legally considered gambling in some countries). Unity doesn't get a cut of that, so they wanted money and they wanted to charge per installation. It sounds reasonable, but of course people immediately realized that if you don't like the company who made a game, you could bankrupt them by just repeatedly installing and uninstalling the game. Even if it's only one cent at a time, automate it and do this enough times and you are a major line item on the balance sheet. Unity walked it back and said they want it to be first-install but per device, but a clever enough person would have no trouble making each install seem like it's a different machine (just lie to the application about hardware IDs or something). Suffice it to say, game developers using the engine revolted and some of them decided they might like to switch to a different engine. So if every request to your service is costing you, there's got to be a way to control that cost.

Even if it's not about preventing a DoS attack or monetary costs, rate limits also prevent someone from scraping all your data. This really happened in early 2021 to the openly-political social media site "Parler". I would certainly forgive you for not being familiar with the site since it was only popular amongst conservative Americans and mostly in the year 2020 before all major companies refused to do business with them. But in any case, according to [Gre21] it was trivially easy for people to download all the content from the site because of two very bad decisions. The first was that all posts had sequential identifiers, so it is trivial to just enumerate all posts (don't do this!). The second is that there was no rate limiting so a single person could then get all the data with a simple script. Okay, you might not be so worried about URLs being completely predictable because that is a very silly thing to implement. Still, it is concerning 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.

The reference to training the machine learning model is an oblique reference to ChatGPT and how it got its data set. Yes, OpenAI did scan the internet to train ChatGPT, although probably not as invasively as how people were pulling the data out of Parler. We can discuss whether it's fair or reasonable for OpenAI to request large amounts of data from others (getting each site to serve up their data at their own execution costs) for OpenAI to then monetize in their model, but that's for another course. Ethics around AI is a big topic indeed—one that you will likely have to wrestle with in your careers.

Dealing With Limits

OK, let's assume that some system you're dealing with has a rate limit, and that we need to address this limitation if we want to increase the performance of the code that is under our control. If the rate limit is super high as compared to our usage of it (e.g., you can do 1000 requests per hour and you are sending 10) then there's no problem right now, but maybe there will be in the future. Probably that is a problem best handled by Future You. Hitting the rate limit regularly, however, is not only frustrating for normal workflows we're trying to run, but could also get us banned as a customer, if it happens too often.

If we start seeing requests rejected with an error that says the rate limit is exceeded, then we are certain it's a problem. It's not always obvious what the limit is, though.

In some scenarios, the documentation tells you the limit, or perhaps there's an API to query it, or API responses for other requests include information about the remaining capacity available to you.

Sometimes, though, the other service does not publish or give information about the rate limit, fearing that it might be exploited or encourage bad behaviour, or because they don't want to commit to anything. Atlassian (they make JIRA, if you're familiar with that) says "REST API rate limits are not published because the computation logic is evolving continuously to maximize reliability and performance for customers" which feels a little bit like they don't

want to commit to specific numbers because then someone might be mad if those numbers aren't met. There is an argument that it does allow more freedom to change the API if there's no commitment to a specific rate. Testing for the limit (spam until we get rejections) might work, but it might also get you banned and, given the above, may give you a number that quickly becomes out of date.

This section would be short indeed if the answer is to shrug our shoulders and say there's nothing we can do. Of course there is! And yet, when I [JZ] was conducting a system design interview for a candidate in 2022, I presented with a scenario with a rate limit. I got the answer that there was nothing that we could do to solve it. That's defeatist and false, as evidenced by the remainder of this section. As you may imagine, we did not make an offer to that candidate. One or more of the things below would have been a much better answer.

Do Less Work. Not the first time you've heard this answer! The first strategy we can imagine is that we should simply reduce the number of our API calls. If the service is pay-per-request, then this also has the benefit of saving money. While I do not imagine that you are intentionally calling the service more than you need, it's possible that there are some redundant calls you could discover through a review. If that's the case, just eliminate them, avoid the rate limit, and reduce latency in your application by avoiding network calls. As said, though, this is likely to be limited in benefit since there's only so much redundancy you are likely to find. Unless we find that a significant number of calls are redundant, we almost certainly need one of the other solutions in this section.

Caching. An easy way to reduce the number of calls to a remote system is to remember the answers that we've previously received. Again, we've covered caching on more than one occasion in the past, so we do not need to discuss the concepts, just where it is applicable and the limitations. The simplest caching strategy would just be used for requests for data and not updates, but we've learned previously about write-through and write-back caches to handle updates. It's possible to make the caching invisible from the point of view of the calling system with something like Redis; a cache that sits outside of the service and can handle requests/responses.

If we do not control the remote system, it may be more difficult to identify when a particular piece of data has changed and a fresh response is needed rather than the cached one. Domain-specific knowledge is helpful here, like with exchange rates. Exchange rates are a price, to some extent: if I'm going to Frankfurt then I need to buy some Euro with my Canadian Dollars. So to buy 100 Euro, the purchase price for me might be $143. Exchange rates vary throughout the day, but perhaps if you request a rate, you get a quote that is valid for 20 minutes. If that's the case, you can add the item to the cache with an expiry of 20 minutes and you can use this value during the validity period without needing to query it again. Other items are harder to manage, of course.

Group Up. The next strategy to consider is whether we can group up requests into one larger request. So instead of, say, five separate requests to update each employee, use one request that updates five employees. The remote API has to allow this; if there is no mass-update API then we're stuck doing them one at a time. The benefit of this may also be limited by how users use the application; if they usually only edit one employee at a time then it does not help because there is nothing else to group the request with. Waiting for other requests might be okay if it's a relatively short wait, but the latency increase does become noticeable for users eventually.

We aren't limited to only grouping related requests, if the remote API will let us combine them. If that's the case, then we can keep a buffer of requests. When we have enough requests ready to go, we can just send them out together. A typical REST-like API does not necessarily give you the ability to do this sort of thing. And again, it also requires something else to group a given request with; if there's nothing else going on, then how long are we willing to wait?

Grouping requests also overlaps with caching if you choose a write-back style cache or another strategy that allows multiple modifications before the eventual call to update the remote system.

Grouping requests may also make it hard to handle what happens 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? It also makes it more likely that such an error occurs, since the logic to build a a larger request is almost certainly significantly more complex than the logic to build a small request. Similarly, the logic to unpack and interpret a larger response is more complex and more prone to bugs. Misunderstanding a response

can easily cause inconsistent data or repeated requests.

These strategies, as with a lot of programming for performance, involve adding implementation complexity in exchange for improved performance. Don't use them when not needed! Simpler implementations are easier to maintain.

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. Assuming time travel hasn't been invented yet and we aren't travelling at relativistic speeds or anything weird like that, the only way for there to be more time is to simply wait. This ultimately means delaying requests to stay under the rate limit. That sounds like the opposite of what we want when we want to go faster, but if we get rate-limit responses the requests are not getting serviced, so we could say that delayed is better than denied.

Outside of the computer context, when there is a more demand for something than capacity, what do we do? That's right, we queue (line up) for it. Here, this means that requests should be added to a queue and a request gets processed when it gets to the front of the queue. Simply controlling the rate at which requests leave the queue is sufficient to ensure that the rate limit is not exceeded. If all requests go through this queue, then it is also a central place to adjust the rate of outgoing requests if the limit changes.

Unsurprisingly, multiple Rust crates do what we want. Below are some examples from the documentation of the `ratelimit` crate[1] version 0.7.1. This is not especially fancy and we could certainly have considered other ones, but it's sufficient for our purposes. The documentation examples cover some implementation details worth talking about. But onward.

```
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();

// 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();
```

[1] https://docs.rs/ratelimit/latest/ratelimit/

```
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
}
```

Enqueueing a request is not always suitable if the user is sitting at the screen and awaiting a response to their request, because it takes a synchronous flow and makes it asynchronous. Rearchitecting the workflows of a user-interactive application may be a larger undertaking than we're willing to do right now in this topic, but it could be a long-term goal for reducing pressure on systems. And even if we can't move all requests to asynchronous, every request that we do make asynchronous takes the pressure off for the ones that need to be synchronous.

For requests that are not urgent, then another option is to schedule the requests for a not-busy time. Applications usually have usage patterns that have busier and less-busy times. So if you know that your application is not used very much overnight, that's a great time to do things that count against your rate limit.

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. Or 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. A final option that you may consider is how to get the other side to raise your limit. Sometimes this just means upgrading to a higher billing tier and you get the higher limit immediately. Sometimes it's something you can simply pay for on top of your existing subscription or agreement. You may also be able to negotiate it with the other side if they're open to that, although you might have to be a sufficiently-important customer to even get in the (Zoom) room for that conversation. Throwing money at the problem can actually work so it's worth considering, but it isn't always a realistic option or is maybe just too expensive. (But don't forget: engineer time is expensive too.)

It Happened Anyway?

Despite our best efforts to reduce it, 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?

It's possible the API provides the answer in the rate-limit response where it says you may try again after a specific period of time. Waiting that long is then the correct thing to do. If we don't know, though, it makes sense to try an exponential backoff strategy. This strategy is to wait a little bit and try again, and if the error occurs, next time wait a little longer than last time, and repeat this procedure until it succeeds.

Exponential backoff is also applicable to unsuccessful requests even if it's not due to rate limiting. If the resource is not available, repeatedly retrying doesn't help. If it's down for maintenance, it could be quite a while before it's back and calling the endpoint 10 times every second doesn't make it come back faster and just wastes effort. 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! And the more failures have occurred, the longer the wait, which gives the service a chance to recover. Eventually, though, you may have to conclude that there's no point in further retries. At that point you can block the thread or return an error, but setting a cap on the maximum retry attempts is reasonable.

Having read a little bit of the source code of the Crossbeam (see Appendix C) implementation of backoff, they don't seem to have any jitter in the request. This is an improvement on the algorithm that prevents all threads or callers from retrying at the exact same time. Let me explain with an example: I once wrote a little program that

tried synchronizing its threads via the database and had an exponential backoff if the thread in question did not successfully lock the item it wanted. I got a lot of warnings in the log about failing to lock, until I added a little randomness to the delay. It makes sense; if two threads fail at time X and they will both retry at time $X + 5$ then they will just fight over the same row. If one thread retries at $X + 9$ and another $X + 7$, they won't conflict.

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 accessing the resource lots of times, you might want something else; something resembling TCP congestion control. See [Aeo19] for details.

References

[Aeo19] Aeoncase. Improve on exponential backoff, 2019. Online; accessed 2020-10-21. URL: `https://www.aeoncase.com/blog/posts/improve-on-exponential-backoff/`.

[Bat23] James Batchelor. Unity clarifies new fee plans amid developer backlash, September 2023. Online; accessed 2023-10-14. URL: `https://www.gamesindustry.biz/unity-clarifies-new-fee-plans-amid-developer-backlash`.

[Gre21] Andy Greenberg. An absurdly basic bug let anyone grab all of parler's data, January 2021. Online; accessed 2023-10-14. URL: `https://www.wired.com/story/parler-hack-data-public-posts-images-video/`.