Workload isolation using shuffle-sharding

Colm MacCárthaigh

Today, Amazon Route 53 hosts many of the world's biggest businesses and most popular websites, but its beginnings are far more humble.

Taking on DNS hosting

Not long after AWS began offering services, AWS customers made clear that they wanted to be able to use our Amazon Simple Storage Service (S3), Amazon CloudFront, and Elastic Load Balancing services at the "root" of their domain, that is, for names like "amazon.com" and not just for names like "www.amazon.com".

That may seem very simple. However, due to a design decision in the DNS protocol, made back in the 1980s, it's harder than it seems. DNS has a feature called CNAME that allows the owner of a domain to offload a part of their domain to another provider to host, but it doesn't work at the root or top level of a domain. To serve our customers' needs, we'd have to actually host our customers' domains. When we host a customer's domain, we can return whatever the current set of IP addresses are for Amazon S3, Amazon CloudFront, or Elastic Load Balancing. These services are constantly expanding and adding IP addresses, so it's not something that customers could easily hard-code in their domain configurations either.

It's no small task to host DNS. If DNS is having problems, an entire business can be offline. However, after we identified the need, we set out to solve it in the way that's typical at Amazon—urgently. We carved out a small team of engineers, and we got to work.

Handling DDOS attacks

Ask any DNS provider what their biggest challenge is and they'll tell you that it's handling distributed denial of service (DDOS) attacks. DNS is built on top of the UDP protocol, which means that DNS requests are spoofable on much of the wild-west internet. Since DNS is also critical infrastructure, this combination makes it an attractive target to unscrupulous actors who try to extort businesses, "booters" who aim to trigger outages for a variety of reasons, and the occasional misguided nuisance maker who doesn't seem to realize they're committing a serious crime with real personal consequences. No matter what the reason, every day there are thousands of DDOS attacks committed against domains.

One approach to mitigate these attacks is to use huge volumes of server capacity. Although it's important to have a good baseline of capacity, this approach doesn't really scale. Every server that a provider adds costs thousands of dollars, but attackers can add more fake clients for pennies if they are using compromised botnets. For providers, adding huge volumes of server capacity is a losing strategy.

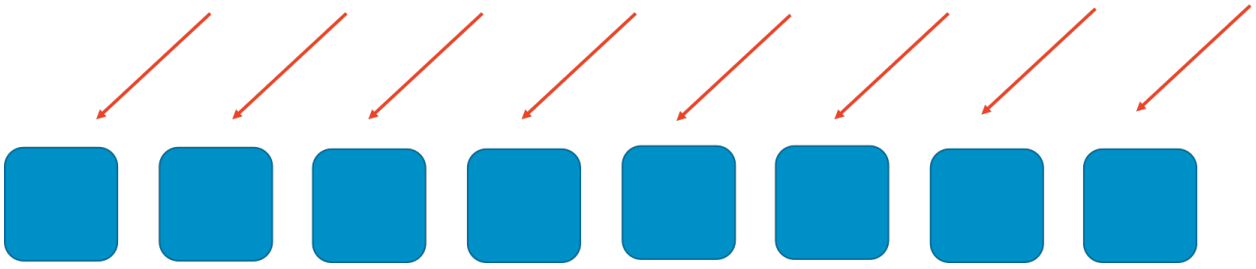
At the time that we built Amazon Route 53, the state of the art for DNS defense was specialized network appliances that could use a variety of tricks to "scrub" traffic at a very high rate. We had many of these appliances at Amazon for our existing in-house DNS services, and we talked to hardware vendors about what else was available. We found out that buying enough appliances to fully cover every single Route 53 domain would cost tens of millions of dollars and add months to our schedule to get them delivered, installed, and operational. That didn't fit with the urgency of our

plans or with our efforts to be frugal, so we never seriously considered them. We needed to find a way to only spend resources defending domains that are actually experiencing an attack. We turned to the old principle that necessity is the mother of invention. Our necessity was to quickly build a world-class, 100 percent uptime DNS service using a modest amount of resources. Our invention was shuffle sharding.

What is shuffle sharding?

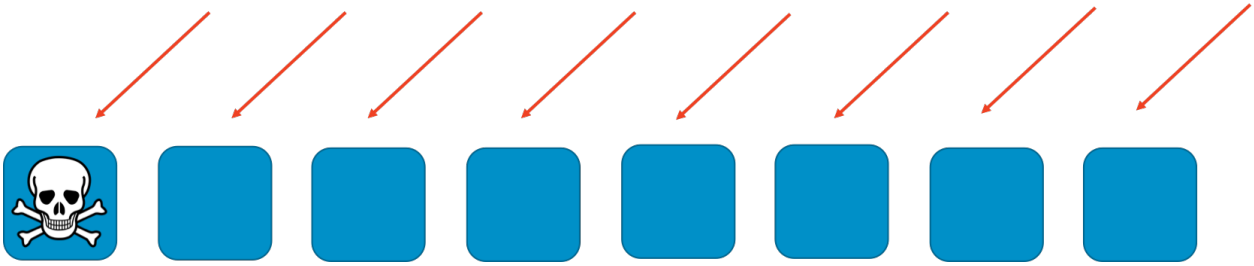
Shuffle sharding is simple, but powerful. It's even more powerful than we first realized. We've used it over and over, and it's become a core pattern that makes it possible for AWS to deliver cost-effective multi-tenant services that give each customer a single-tenant experience.

To see how shuffle sharding works, first consider how a system can be made more scalable and resilient through ordinary sharding. Imagine a horizontally scalable system or service that is made up of eight workers. The following image illustrates workers and their requests. The workers could be



servers, or queues, or databases, whatever the "thing" it is that makes up your system.

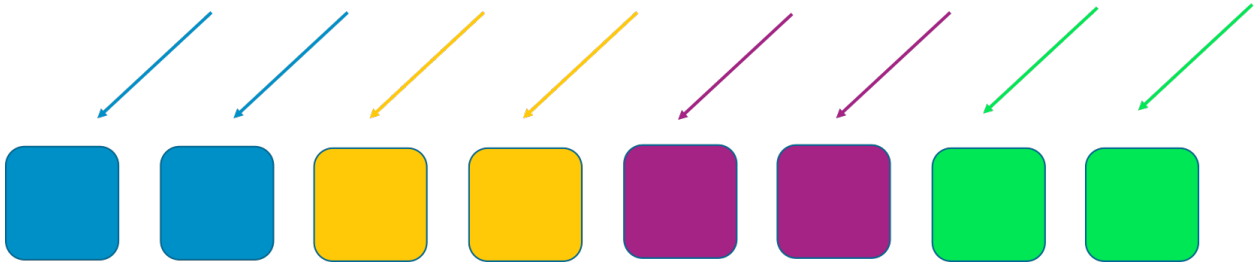
Without any sharding, the fleet of workers handles all of the work. Each worker has to be able to handle any request. This is great for efficiency and redundancy. If a worker fails, the other seven can absorb the work, so relatively little slack capacity is needed in the system. However, a big problem crops up if failures can be triggered by a particular kind of request, or by a flood of requests, such as a DDOS attack. The following two images show the progression of such an attack.



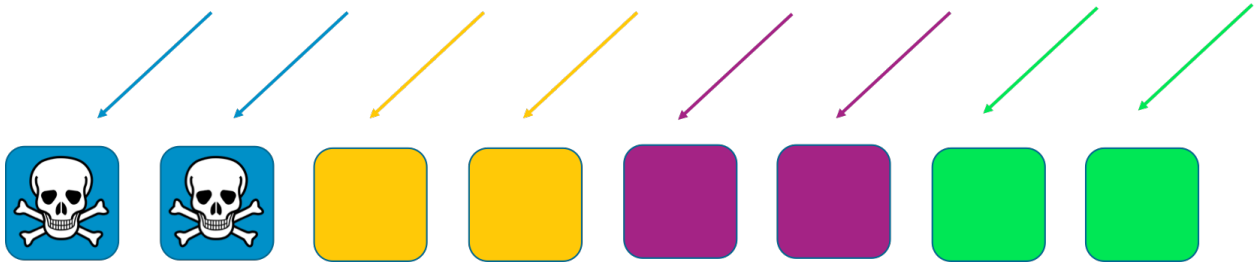
The problem will take out the first worker impacted, but then proceed to cascade through the other workers as the remaining workers take over. The problem can very quickly take out all of the workers, and the entire service.

The scope of impact for this kind of failure is “everything and everyone.” The whole service goes down. Every customer is impacted. As we say in availability engineering: It’s not optimal.

With regular sharding we can do better. If we divide the fleet into 4 shards of workers, we can trade efficiency for scope of impact. The following two images show how sharding can limit the impact of a DDOS attack.



In this example, each shard has two workers. We split resources, such as customer domains, across the shards. We still have redundancy, but because there are only two workers per shard, we have to keep more slack capacity in the system to handle any failures. In return, the scope of impact is reduced considerably.

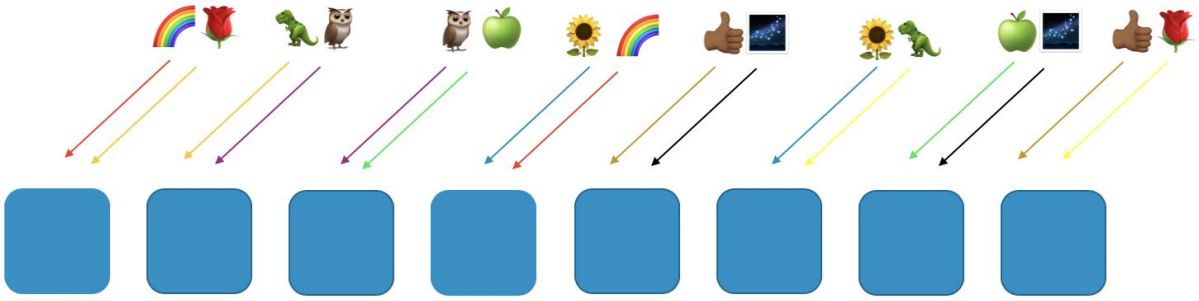


In this sharding world, the scope of impact is reduced by the number of shards. Here with four shards, if a customer experiences a problem, then the shard hosting them might be impacted, as well as all of the other customers on that shard. However, that shard represents just one quarter of the overall service. A 25 percent impact is much better than a 100 percent impact. With shuffle sharding, we can do exponentially better again.

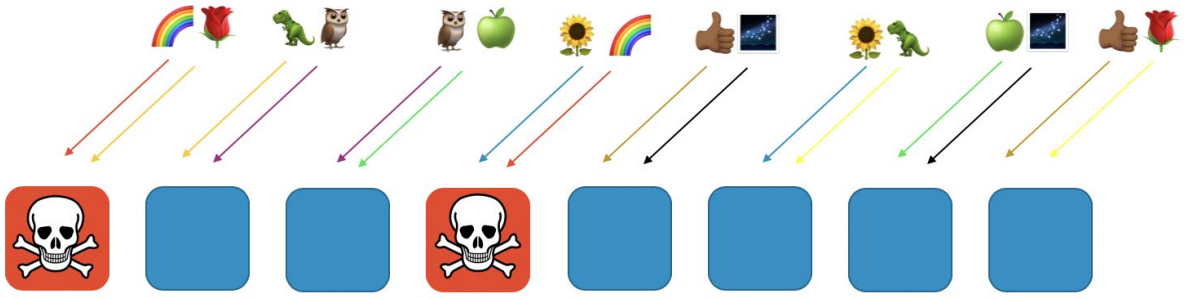
With shuffle sharding we create virtual shards of two workers each, and we assign our customers or resources, or whatever we want to isolate, to one of those virtual shards.

The following image shows an example shuffle sharding layout with eight workers and eight customers, who are each assigned to two workers. Normally we'd have many more customers than workers, but it's easier to follow along if we keep things smaller. We'll focus on two customers—the rainbow customer and the rose customer.

In our example, we assign the rainbow customer to the first worker and to the fourth worker. The combination of those two workers makes up that customer's shuffle shard. Other customers will go to different virtual shards, with their own mix of two workers. For example, the rose customer is also assigned to the first worker, but its other worker is the eighth worker.



If the rainbow customer assigned to workers one and four has a problem (such as a poisonous request, or a flood of requests), that problem will impact that virtual shard, but it won't fully impact any other shuffle shard. In fact, at most one of another shuffle shard's workers will be affected. If the requestors are fault tolerant and can work around this (with retries for example), service can continue uninterrupted for the customers or resources on the remaining shards, as the following image illustrates.



Put another way, while all of the workers serving rainbow might be experiencing a problem or an attack, the other workers aren't affected at all. For customers, that means that even though the rose customer and the sunflower customer each share a worker with the rainbow, they aren't impacted. The rose customer can get service from worker eight, and the sunflower can get service from worker six, as seen in the following image.

When a problem happens, we can still lose a quarter of the whole service, but the way that customers or resources are assigned means that the scope of impact with shuffle sharding is considerably better. With eight workers, there are 28 unique combinations of two workers, which means that there are 28 possible shuffle shards. If we have hundreds or more of customers, and we assign each customer to a shuffle shard, then the scope of impact due to a problem is just $1/28^{th}$. That's 7 times better than regular sharding.

It's very exciting to see that the numbers get exponentially better the more workers and customers you have. Most scaling challenges get harder in those dimensions, but shuffle sharding gets more effective. In fact, with enough workers, there can be more shuffle shards then there are customers, and each customer can be isolated.

Amazon Route 53 and shuffle sharding

How does all of this help Amazon Route 53? With Route 53, we decided to arrange our capacity into a total of 2048 virtual name servers. These servers are virtual because they don't correspond to the physical servers hosting Route 53. We can move them around to help manage capacity. We then assign every customer domain to a shuffle shard of four virtual name servers. With those numbers, there are a staggering 730 billion possible shuffle shards. We have so many possible shuffle shards that we can assign a unique shuffle shard to every domain. Actually, we can go further, and ensure that no customer domain will ever share more than two virtual name servers with any other customer domain.

The results are amazing. If a customer domain is targeted for a DDOS attack, the four virtual name servers assigned to that domain will spike in traffic, but no other customer's domain will notice. We're not resigned to the targeted customer having a bad day. Shuffle sharding means that we can identify and isolate the targeted customer to special dedicated attack capacity. In addition, we've also developed our own proprietary layer of AWS Shield traffic scrubbers. However, shuffle sharding makes a massive difference in ensuring that the overall Route 53 customer experience is seamless, even while these events are happening.

Conclusion

We've gone on to embed shuffle sharding in many of our other systems. In addition, we've come up with refinements, such as recursive shuffle sharding, where we shard items at multiple layers, thus

isolating a customer's customer. Shuffle sharding is enormously adaptable. It's a smart way to arrange existing resources. It also usually comes at no additional cost, so it's a great improvement that frugality and economy can drive.

If you're interested in using shuffle sharding yourself, check out our open source Route 53 Infima library. This library includes several different implementations of shuffle sharding that can be used for assigning or arranging resources.