The role of a reverse proxy to protect your application against slow clients

brianstorti.com/the-role-of-a-reverse-proxy-to-protect-your-application-against-slow-clients

January 6, 2015

When you are running an application server that uses a forking model, slow clients can make your application simply stop handling new requests. Slow clients can be just users with a slow connection sending a large request, or an attacker, being slow on purpose. I'll try to explain what these slow clients are and how a reverse proxy can be used to protect your application server against them.

But first we need to understand what a forking model is.

The forking model

Saying that an application server uses a forking model, simply put, means that it will spawn processes to handle new requests. As long as its concurrency strategy is based on using new processes to handle more requests, we can consider it to be using a forking model.

A well known application server that follows this strategy is Unicorn.

The problem with slow clients

Let's try to imagine this scenario: There is an user trying to access you application. This user has a really terrible connection, maybe he/she is using a mobile network (or a 56k modem, who knows?), and this user is trying to send you a large request, a 5MB picture, for instance.

When your application server receives this requests, it spawns a new process to handle it, and start receiving that large request. The process will be bottlenecked by the speed of the client connection, and it will be blocked until the slow client finishes sending that large picture. Being blocked means that this worker process can not handle any other request in the meantime, it's just there, idle, waiting to receive the entire request so it can start really processing it.
The same problem happens when it needs to send back a response. If the client is slow to receive it, the process will keep blocked, not being able to handle other requests.

When all of your worker processes are busy (maybe just because they are blocked by slow clients), your application stops receiving any new requests. That's not good.

Buffering reverse proxies for the rescue

A reverse proxy (like Nginx) seats in front of your application server (say, Unicorn), and can offer a sort of buffering system.

This buffering reverse proxy can handle an “unlimited” number of requests, and is not affected by slow clients.
Nginx, for instance, uses a non-blocking Evented I/O model (rather than the I/O blocking forking model that Unicorn uses), which means that, when it receives a new request, it will perform a read call (I/O operation), and will not be blocked waiting for a response, being immediately available to handle new requests. When the read operation finishes, the operational system will send an event notification, and the appropriate event handler can be called (passing the request to the application server, for instance).

The scenario above, with a buffering reverse proxy, would be something like this: The slow client makes a large request. The buffering reverse proxy will wait until it gets the entire request, then it will pass this request to the application server, which will just process it and deliver the response back to the reverse proxy, being free to receive new requests. The reverse proxy then will send this response back to the slow client.
It doesn’t really matter much that it will take a long time to receive and deliver the requests/responses to these clients, as the reverse proxy will not be blocked by these I/O operations (due to its concurrency model nature).

Now the application server processes are responsible just for processing the request, not being blocked by these slow clients anymore.

The conclusion here is that, if you are using an application server that is blocked by I/O operations, it’s a pretty good idea to put a reverse proxy in front of it, that can handle this kind of situations (and, possibly, do a lot more).

Interested in learning Kubernetes?

Get fresh articles in your inbox